# Table of Contents

1 Introduction ................................................................. 1
   1.1 Overview of Gauche .................................................... 1
   1.2 Notations .................................................................... 2
       1.2.1 Entry format ....................................................... 2
       1.2.2 Names and namespaces ........................................ 4

2 Concepts ........................................................................ 5
   2.1 Standard conformance .................................................. 5
   2.2 Multibyte strings ........................................................... 13
   2.3 Multibyte scripts ............................................................ 14
   2.4 Case-sensitivity ............................................................. 15
   2.5 Integrated object system ............................................... 15
   2.6 Module system ............................................................. 16
   2.7 Compilation ................................................................. 17

3 Programming in Gauche .................................................. 19
   3.1 Invoking Gosh .............................................................. 19
   3.2 Interactive development ............................................... 24
       3.2.1 Working in REPL ................................................. 24
       3.2.2 Input editing ....................................................... 29
   3.3 Writing Scheme scripts ................................................ 31
   3.4 Debugging ................................................................. 33
   3.5 Using platform-dependent features .................................. 34
   3.6 Profiling and tuning ..................................................... 36
       3.6.1 Using profiler ...................................................... 36
       3.6.2 Performance tips .................................................. 37
   3.7 Writing Gauche modules .............................................. 38
   3.8 Using extension packages ............................................. 39
   3.9 Building standalone executables .................................... 40

4 Core syntax ..................................................................... 43
   4.1 Lexical structure ........................................................... 43
       4.1.1 Sharp syntax ....................................................... 44
       4.1.2 Hash-bang token .................................................. 46
   4.2 Literals ...................................................................... 46
   4.3 Making procedures ..................................................... 47
   4.4 Assignments ............................................................... 52
   4.5 Conditionals ............................................................... 55
   4.6 Binding constructs ...................................................... 59
   4.7 Grouping ................................................................... 62
   4.8 Iteration .................................................................... 63
   4.9 Quasiquotation ............................................................ 66
   4.10 Definitions ............................................................... 68
       4.10.1 Into the Scheme-Verse ......................................... 72
   4.11 Inclusions ................................................................. 74
   4.12 Feature conditional ..................................................... 75
   4.13 Modules ................................................................... 78
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13.1</td>
<td>Module semantics</td>
<td>78</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Modules and libraries</td>
<td>80</td>
</tr>
<tr>
<td>4.13.3</td>
<td>Defining and selecting modules</td>
<td>81</td>
</tr>
<tr>
<td>4.13.4</td>
<td>Using modules</td>
<td>81</td>
</tr>
<tr>
<td>4.13.5</td>
<td>Module inheritance</td>
<td>83</td>
</tr>
<tr>
<td>4.13.6</td>
<td>Module introspection</td>
<td>84</td>
</tr>
<tr>
<td>4.13.7</td>
<td>Predefined modules</td>
<td>85</td>
</tr>
<tr>
<td>5</td>
<td>Macros</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Why hygienic?</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>Hygienic macros</td>
<td>90</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Syntax-rules macro transformer</td>
<td>91</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Explicit-renaming macro transformer</td>
<td>93</td>
</tr>
<tr>
<td>5.3</td>
<td>Traditional macros</td>
<td>97</td>
</tr>
<tr>
<td>5.4</td>
<td>Hybrid macros</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>Identifiers</td>
<td>98</td>
</tr>
<tr>
<td>5.6</td>
<td>Debugging macros</td>
<td>100</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Tracing macro expansion</td>
<td>100</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Expanding macros manually</td>
<td>102</td>
</tr>
<tr>
<td>5.7</td>
<td>Macro utilities</td>
<td>104</td>
</tr>
<tr>
<td>6</td>
<td>Core library</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>Types and classes</td>
<td>105</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Prescriptive and descriptive types</td>
<td>105</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Generic type predicates</td>
<td>105</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Predefined classes</td>
<td>106</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Type expressions and type constructors</td>
<td>107</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Native types</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>Equality and comparison</td>
<td>110</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Equality</td>
<td>110</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Comparison</td>
<td>112</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Hashing</td>
<td>113</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Basic comparators</td>
<td>116</td>
</tr>
<tr>
<td>6.2.4.1</td>
<td>Comparator class and constructors</td>
<td>116</td>
</tr>
<tr>
<td>6.2.4.2</td>
<td>Comparator predicates and accessors</td>
<td>118</td>
</tr>
<tr>
<td>6.2.4.3</td>
<td>Predefined comparators</td>
<td>119</td>
</tr>
<tr>
<td>6.2.4.4</td>
<td>Combining comparators</td>
<td>121</td>
</tr>
<tr>
<td>6.3</td>
<td>Numbers</td>
<td>122</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Number classes</td>
<td>122</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Numerical predicates</td>
<td>123</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Numerical comparison</td>
<td>125</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Arithmetics</td>
<td>126</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Numerical conversions</td>
<td>134</td>
</tr>
<tr>
<td>6.3.6</td>
<td>Basic bitwise operations</td>
<td>136</td>
</tr>
<tr>
<td>6.3.7</td>
<td>Endianness</td>
<td>138</td>
</tr>
<tr>
<td>6.4</td>
<td>Booleans</td>
<td>139</td>
</tr>
<tr>
<td>6.5</td>
<td>Undefined values</td>
<td>140</td>
</tr>
<tr>
<td>6.6</td>
<td>Pairs and lists</td>
<td>140</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Pair and null class</td>
<td>141</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Mutable and immutable pairs</td>
<td>141</td>
</tr>
<tr>
<td>6.6.3</td>
<td>List predicates</td>
<td>141</td>
</tr>
<tr>
<td>6.6.4</td>
<td>List constructors</td>
<td>142</td>
</tr>
<tr>
<td>6.6.5</td>
<td>List accessors and modifiers</td>
<td>143</td>
</tr>
</tbody>
</table>
6.24.7.3 Handling signals ...................................................... 309
6.24.7.4 Masking and waiting signals ........................................ 311
6.24.7.5 Signals and threads ................................................... 312
6.24.8 System inquiry .......................................................... 313
6.24.9 Time .................................................................... 316
6.24.10 Process management ..................................................... 318
6.24.11 I/O multiplexing .......................................................... 321
6.24.12 Garbage collection ........................................................ 323
6.24.13 Memory mapping .......................................................... 323
6.24.14 Miscellaneous system calls ............................................ 324
6.25 Development helper API .................................................. 326
6.25.1 Debugging aid ............................................................. 326
6.25.2 Profiler API ................................................................. 327

7 Object system .................................................................. 329
7.1 Introduction to the object system ........................................... 329
7.2 Class ........................................................................ 336
  7.2.1 Defining class ............................................................ 336
  7.2.2 Inheritance ................................................................. 338
  7.2.3 Class object ............................................................... 340
  7.2.4 Slot definition object ................................................... 341
  7.2.5 Class redefinition ........................................................ 342
  7.2.6 Class definition examples .............................................. 343
7.3 Instance ...................................................................... 345
  7.3.1 Creating instance .......................................................... 345
  7.3.2 Accessing instance ....................................................... 346
  7.3.3 Changing classes ........................................................ 347
7.4 Generic function and method ............................................... 349
7.5 Metaobject protocol .......................................................... 351
  7.5.1 Class instantiation ........................................................ 351
  7.5.2 Customizing slot access ............................................... 354
  7.5.3 Method instantiation ..................................................... 357
  7.5.4 Customizing method application ..................................... 357
  7.5.5 Customizing class redefinition ....................................... 358

8 Library modules - Overview .............................................. 359
8.1 Finding libraries you need ................................................... 359
  8.1.1 Library directory - data containers ................................. 360
  8.1.2 Library directory - string and character .......................... 361
  8.1.3 Library directory - data exchange .................................. 361
  8.1.4 Library directory - files .............................................. 362
  8.1.5 Library directory - processes and threads ....................... 362
  8.1.6 Library directory - networking ....................................... 362
  8.1.7 Library directory - input and output ............................... 362
  8.1.8 Library directory - time .............................................. 363
  8.1.9 Library directory - bits and bytes .................................. 363
8.2 Naming convention of libraries .......................................... 363
8.3 Obsolete and superseded modules ........................................ 364
9  Library modules - Gauche extensions .......................... 366
   9.1  gauche.array - Arrays ........................................ 366
   9.2  gauche.base - Importing gauche built-ins ................... 373
   9.3  gauche.bitvector - Bitvector utilities ...................... 374
   9.4  gauche.cgen - Generating C code ............................. 382
       9.4.1  Generating C source files ............................. 383
       9.4.2  Generating Scheme literals ............................ 387
       9.4.3  Conversions between Scheme and C ...................... 388
       9.4.4  CiSE - C in S expression ............................... 390
           9.4.4.1  CiSE overview .................................... 391
           9.4.4.2  CiSE syntax ...................................... 391
           9.4.4.3  CiSE procedures ................................. 395
       9.4.5  Stub generation ......................................... 396
   9.5  gauche.charconv - Character Code Conversion ................. 399
       9.5.1  Supported character encoding schemes .................. 399
       9.5.2  Autodetecting the encoding scheme ..................... 402
       9.5.3  Conversion ports ........................................ 402
   9.6  gauche.collection - Collection framework .................... 404
       9.6.1  Mapping over collection ................................ 405
       9.6.2  Selection and searching in collection .................. 408
       9.6.3  Miscellaneous operations on collection ................. 409
       9.6.4  Fundamental iterator creators .......................... 410
       9.6.5  Implementing collections ............................... 412
   9.7  gauche.config - Configuration parameters ..................... 413
   9.8  gauche.configure - Generating build files ................... 413
       9.8.1  Structure of configure script and build files ......... 413
       9.8.2  Configure API .......................................... 414
   9.9  gauche.connection - Connection framework .................... 426
   9.10 gauche.dictionary - Dictionary framework ..................... 428
       9.10.1  Generic functions for dictionaries .................... 428
       9.10.2  Generic dictionaries .................................. 430
   9.11  gauche.fcntl - Low-level file operations .................... 432
   9.12  gauche.generator - Generators ............................... 436
       9.12.1  Generator constructors ................................ 436
       9.12.2  Generator operations .................................. 440
       9.12.3  Generator consumers ................................... 445
   9.13  gauche.hook - Hooks ......................................... 447
   9.14  gauche.interactive - Utilities for interactive session .... 448
   9.15  gauche.lazy - Lazy sequence utilities ....................... 451
       9.15.1  Lazy sequence constructors ............................ 451
       9.15.2  Lazy sequence operations .............................. 452
       9.15.3  Lazy sequence with positions .......................... 454
   9.16  gauche.listener - Listener .................................. 455
   9.17  gauche.logger - User-level logging .......................... 458
   9.18  gauche.mop.instance-pool - Instance pools ................... 460
   9.19  gauche.mop.propagate - Propagating slot access ............ 461
   9.20  gauche.mop.singleton - Singleton ............................ 463
   9.21  gauche.mop.validator - Slot with validator .................. 463
   9.22  gauche.net - Networking .................................... 464
       9.22.1  Socket address ........................................ 465
       9.22.2  High-level network functions .......................... 467
       9.22.3  Low-level socket interface ............................ 470
       9.22.4  Netdb interface ....................................... 475
   9.23  gauche.package - Package metainformation .................... 477
9.21  gauche.parameter - Parameters (legacy) ........................................... 480
9.25  gauche.parseopt - Parsing command-line options ............................ 480
9.26  gauche.partcont - Partial continuations ......................................... 485
9.27  gauche.process - High-level process interface .................................... 488
   9.27.1  Running subprocess ................................................................... 488
   9.27.2  Running process pipeline ............................................................ 494
   9.27.3  Process object ........................................................................... 494
   9.27.4  Process ports ............................................................................. 498
   9.27.5  Process connection ................................................................. 501
9.28  gauche.record - Record types ............................................................. 502
   9.28.1  Introduction .............................................................................. 502
   9.28.2  Syntactic Layer ......................................................................... 503
   9.28.3  Inspection layer ......................................................................... 505
   9.28.4  Procedural layer ....................................................................... 506
   9.28.5  Pseudo record types .................................................................. 507
9.29  gauche.reload - Reloading modules ........................................................ 508
9.30  gauche.selector - Simple dispatcher ................................................... 509
9.31  gauche.sequence - Sequence framework ............................................. 510
   9.31.1  Fundamental sequence accessors .............................................. 510
   9.31.2  Slicing sequence ...................................................................... 511
   9.31.3  Mapping over sequences ............................................................ 511
   9.31.4  Other operations over sequences .............................................. 512
   9.31.5  Implementing sequence .............................................................. 517
9.32  gauche.syslog - Syslog ..................................................................... 517
9.33  gauche.termios - Terminal control ....................................................... 518
   9.33.1  Posix termios interface ............................................................... 518
   9.33.2  Common high-level terminal control ......................................... 521
9.34  gauche.test - Unit Testing .................................................................. 521
   9.34.1  Structuring a test file .................................................................. 522
   9.34.2  Individual test ........................................................................... 523
   9.34.3  Incorporating external tests ....................................................... 526
   9.34.4  Better test failure reporting ....................................................... 527
   9.34.5  Quasi-static checks .................................................................... 529
9.35  gauche.threads - Threads ................................................................. 530
   9.35.1  Thread programming tips ........................................................... 530
   9.35.2  Thread procedures ................................................................... 532
   9.35.3  Thread local storage ................................................................. 535
   9.35.4  Synchronization primitives ......................................................... 536
      9.35.4.1  Mutex ............................................................................ 536
      9.35.4.2  Condition variable .............................................................. 539
      9.35.4.3  Atom .............................................................................. 540
      9.35.4.4  Semaphore ...................................................................... 541
      9.35.4.5  Latch ............................................................................... 542
      9.35.4.6  Barrier ............................................................................. 542
   9.35.5  Thread exceptions ..................................................................... 543
9.36  gauche.time - Measure timings ............................................................. 546
9.37  gauche.unicode - Unicode utilities ....................................................... 549
   9.37.1  Unicode transfer encodings ....................................................... 550
   9.37.2  Unicode text segmentation ....................................................... 553
   9.37.3  Full string case conversion ....................................................... 554
   9.37.4  East asian width property ......................................................... 555
9.38  gauche.uvector - Uniform vector library ............................................... 556
   9.38.1  Uvector basic operations ........................................................... 557
   9.38.2  Uvector conversion operations ................................................... 563
10 Library modules - R7RS standard libraries .......... 580
  10.1 R7RS integration ........................................... 580
      10.1.1 Traveling between two worlds back and forth ....... 580
      10.1.2 Three import forms .................................. 582
  10.2 R7RS small language ........................................ 584
      10.2.1 R7RS library form ....................................... 584
      10.2.2 scheme.base - R7RS base library ............... 585
      10.2.3 scheme.case-lambda - R7RS case-lambda ....... 587
      10.2.4 scheme.char - R7RS char library ................. 587
      10.2.5 scheme.complex - R7RS complex numbers ........... 588
      10.2.6 scheme.cxr - R7RS cxr accessor ... 588
      10.2.7 scheme.eval - R7RS eval ................................... 588
      10.2.8 scheme.file - R7RS file library ................. 589
      10.2.9 scheme.inexact - R7RS inexact numbers ........... 589
      10.2.10 scheme.lazy - R7RS lazy evaluation ............... 590
      10.2.11 scheme.load - R7RS load ............................... 590
      10.2.12 scheme.process-context - R7RS process context .... 590
      10.2.13 scheme.read - R7RS read ................................ 591
      10.2.14 scheme.repl - R7RS repl .............................. 591
      10.2.15 scheme.time - R7RS time .............................. 591
      10.2.16 scheme.write - R7RS write ............................ 592
      10.2.17 scheme.r5rs - R5RS compatibility ............... 592
  10.3 R7RS large .................................................. 593
      10.3.1 scheme.list - R7RS lists ............................... 593
      10.3.2 scheme.vector - R7RS vectors ......................... 597
      10.3.3 scheme.vector.0 - R7RS uniform vectors ........... 602
      10.3.4 scheme.sort - R7RS sort ............................... 602
      10.3.5 scheme.set - R7RS sets ................................. 606
      10.3.6 scheme.charset - R7RS character sets ............... 614
          10.3.6.1 Character-set constructors .............. 614
          10.3.6.2 Character-set comparison .............. 615
          10.3.6.3 Character-set iteration .............. 615
          10.3.6.4 Character-set query .............. 616
          10.3.6.5 Character-set algebra .............. 617
      10.3.7 scheme.hash-table - R7RS hash tables ............... 618
      10.3.8 scheme.i1list - R7RS immutable lists ............. 621
      10.3.9 scheme.r1list - R7RS random-access lists .......... 622
      10.3.10 scheme.ideque - R7RS immutable deques .......... 623
      10.3.11 scheme.text - R7RS immutable texts ............. 627
      10.3.12 scheme.generator - R7RS generators .......... 631
      10.3.13 scheme.lseq - R7RS lazy sequences ............... 633
      10.3.14 scheme.stream - R7RS stream ....................... 635
      10.3.15 scheme.box - R7RS boxes .............................. 636
      10.3.16 scheme.list-queue - R7RS list queues .............. 636
      10.3.17 scheme.ephemeron - R7RS ephemeron ............... 639
      10.3.18 scheme.comparator - R7RS comparators .......... 640
      10.3.19 scheme.regex - R7RS regular expressions ........ 640
      10.3.20 scheme.mapping - R7RS mappings ................... 652
### 11 Library modules - SRFIs

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Testing availability of SRFIs</td>
<td>689</td>
</tr>
<tr>
<td>11.2</td>
<td>SRFIs that have become R7RS-large</td>
<td>689</td>
</tr>
<tr>
<td>11.3</td>
<td>srfi.4 - Homogeneous vectors</td>
<td>691</td>
</tr>
<tr>
<td>11.4</td>
<td>srfi.5 - A compatible let form with signatures and rest arguments</td>
<td>691</td>
</tr>
<tr>
<td>11.5</td>
<td>srfi.7 - Feature-based program configuration language</td>
<td>691</td>
</tr>
<tr>
<td>11.6</td>
<td>srfi.13 - String library</td>
<td>692</td>
</tr>
<tr>
<td>11.7</td>
<td>srfi.19 - Time data types and procedures</td>
<td>700</td>
</tr>
<tr>
<td>11.8</td>
<td>srfi.27 - Sources of Random Bits</td>
<td>706</td>
</tr>
<tr>
<td>11.9</td>
<td>srfi.29 - Localization</td>
<td>707</td>
</tr>
<tr>
<td>11.10</td>
<td>srfi.37 - args-fold: a program argument processor</td>
<td>709</td>
</tr>
<tr>
<td>11.11</td>
<td>srfi.42 - Eager comprehensions</td>
<td>711</td>
</tr>
<tr>
<td>11.12</td>
<td>srfi.43 - Vector library (legacy)</td>
<td>717</td>
</tr>
<tr>
<td>11.13</td>
<td>srfi.55 - Requiring extensions</td>
<td>718</td>
</tr>
<tr>
<td>11.14</td>
<td>srfi.60 - Integers as bits</td>
<td>718</td>
</tr>
<tr>
<td>11.15</td>
<td>srfi.64 - A Scheme API for test suites</td>
<td>720</td>
</tr>
<tr>
<td>11.15.1</td>
<td>Test API</td>
<td>720</td>
</tr>
<tr>
<td>11.15.2</td>
<td>Test runner</td>
<td>720</td>
</tr>
<tr>
<td>11.16</td>
<td>srfi.66 - Octet vectors</td>
<td>720</td>
</tr>
<tr>
<td>11.17</td>
<td>srfi.69 - Basic hash tables</td>
<td>721</td>
</tr>
<tr>
<td>11.18</td>
<td>srfi.74 - Octet-addressed binary blocks</td>
<td>723</td>
</tr>
<tr>
<td>11.19</td>
<td>srfi.78 - Lightweight testing</td>
<td>725</td>
</tr>
<tr>
<td>11.20</td>
<td>srfi.98 - Accessing environment variables</td>
<td>727</td>
</tr>
<tr>
<td>11.21</td>
<td>srfi.101 - Purely functional random-access pairs and lists</td>
<td>727</td>
</tr>
<tr>
<td>11.22</td>
<td>srfi.106 - Basic socket interface</td>
<td>727</td>
</tr>
<tr>
<td>11.23</td>
<td>srfi.112 - Environment inquiry</td>
<td>730</td>
</tr>
</tbody>
</table>
11.24 srfl.114 - Comparators ................................................... 731
11.25 srfl.118 - Simple adjustable-size strings .......................... 736
11.26 srfl.120 - Timer APIs .................................................... 736
11.27 srfl.129 - Titlecase procedures ...................................... 738
11.28 srfl.130 - Cursor-based string library ............................... 738
11.29 srfl.152 - String library (reduced) ................................... 740
11.30 srfl.154 - First-class dynamic extents .............................. 742
11.31 srfl.160 - Homogeneous numeric vector libraries ................ 743
11.32 srfl.162 - Comparator sublibrary ................................. 744
11.33 srfl.170 - POSIX API .................................................. 744
11.34 srfl.172 - Two safer subsets of R7RS .................... 750
11.35 srfl.173 - Hooks (srfl) .............................................. 751
11.36 srfl.174 - POSIX timespecs .......................................... 752
11.37 srfl.175 - ASCII character library .................................. 753
11.38 srfl.178 - Bitvector library ...................................... 756
11.39 srfl.180 - JSON ...................................................... 757
11.40 srfl.181 - Custom ports ........................................... 760
11.40.1 Creating custom ports ........................................ 760
11.40.2 Transcoded ports .................................................. 761
11.41 srfl.185 - Linear adjustable-length strings .................... 764
11.42 srfl.189 - Maybe and Either: optional container types .......... 765
11.42.1 Types and predicates ........................................... 765
11.42.2 Constructors ...................................................... 766
11.42.3 Accessors .......................................................... 766
11.42.4 Sequence operations ............................................. 767
11.42.5 Protocol converters ............................................. 769
11.42.6 Syntactic utilities ................................................ 771
11.42.7 Trivalent logic .................................................... 772
11.43 srfl.192 - Port positioning .......................................... 773
11.44 srfl.193 - Command line ........................................... 773
11.45 srfl.196 - Range objects ........................................... 774
11.46 srfl.197 - Pipeline operators ....................................... 774
11.47 srfl.207 - String-notated bytevectors .............................. 776
11.48 srfl.209 - Enums and enum sets ................................... 779
11.49 srfl.210 - Procedures and syntax for multiple values ............ 784
11.50 srfl.216 - SICP prerequisites ..................................... 786
11.51 srfl.217 - Integer sets .............................................. 787
11.52 srfl.219 - Define higher-order lambda ............................ 792
11.53 srfl.221 - Generator/accumulator sub-library ..................... 792
11.54 srfl.222 - Compound objects ....................................... 794
11.55 srfl.226 - Control features .......................................... 796
11.56 srfl.227 - Optional arguments ..................................... 800
11.57 srfl.228 - Composing comparators ................................ 801
11.58 srfl.229 - Tagged procedures ...................................... 802
11.59 srfl.232 - Flexible curried procedures ............................ 803
11.60 srfl.235 - Combinators (SRFI) .................................... 804
11.61 srfl.239 - Destructuring lists ...................................... 807

12 Library modules - Utilities ............................................. 808

12.1 binary.io - Binary I/O ................................................. 808
12.2 binary.pack - Packing binary data .................................. 811
12.3 compat.chibi-test - Running Chibi-scheme test suite .......... 814
12.4 compat.norational - Rational-less arithmetic ................... 814
12.5 compat.real-elementary-functions -
Backward-compatible real elementary functions ........................................... 815
12.6 control.cseq - Concurrent sequences .................................................. 816
12.7 control.future - Futures ........................................................................ 816
12.8 control.job - A common job descriptor for control modules ...................... 817
12.9 control.plumbing - Plumbing ports ......................................................... 819
12.10 control.pmap - Parallel map ................................................................... 822
12.11 control.scheduler - Scheduler ................................................................. 824
12.12 control.thread-pool - Thread pools ......................................................... 826
12.13 crypt.bcrypt - Password hashing ............................................................ 827
12.14 data.cache - Cache ................................................................................. 828
12.15 data.heap - Heap ................................................................................... 831
12.16 data.ideal - Immutable deques ................................................................. 833
12.17 data.imap - Immutable map ...................................................................... 834
12.18 data.priority-map - Priority map ............................................................. 836
12.19 data.queue - Queue .................................................................................. 837
12.20 data.random - Random data generators .................................................. 841
12.21 data.range - Range .................................................................................. 845
12.22 data.ring-buffer - Ring buffer ................................................................. 849
12.23 data.skew-list - Skew binary random-access lists .................................... 851
12.24 data.sparse - Sparse data containers ...................................................... 853
  12.24.1 Sparse vectors .................................................................................... 854
  12.24.2 Sparse matrixes .................................................................................. 856
  12.24.3 Sparse tables ...................................................................................... 858
12.25 data.trie - Trie ......................................................................................... 859
12.26 data.ulid - Universally unique lexicographically sortable identifier ........ 863
12.27 dbi - Database independent access layer ............................................... 865
  12.27.1 DBI user API ...................................................................................... 865
  12.27.2 Writing drivers for DBI ....................................................................... 869
12.28 dbm - Generic DBM interface ................................................................. 871
  12.28.1 Opening and closing a dbm database ................................................. 872
  12.28.2 Accessing a dbm database .................................................................. 873
  12.28.3 Iterating on a dbm database .............................................................. 874
  12.28.4 Managing dbm database instance ..................................................... 874
  12.28.5 Dumping and restoring dbm database ............................................... 875
  12.28.6 Writing a dbm implementation ......................................................... 875
12.29 dbm.fsdbm - File-system dbm ................................................................. 876
12.30 dbm.gdbm - GDBM interface ................................................................. 876
12.31 dbm.ndbm - NDBM interface ................................................................. 878
12.32 dbm.odbm - Original DBM interface ...................................................... 879
12.33 file.filter - Filtering file content ......................................................... 880
12.34 file.util - Filesystem utilities ............................................................... 881
  12.34.1 Directory utilities .............................................................................. 881
  12.34.2 Pathname utilities ............................................................................. 884
  12.34.3 File attribute utilities ...................................................................... 886
  12.34.4 File operations .................................................................................. 888
  12.34.5 Temporary files and directories ....................................................... 890
  12.34.6 Lock files ......................................................................................... 891
12.35 math.const - Mathematical constants .................................................. 893
12.36 math.mt-random - Mersenne Twister Random number generator ........ 893
12.37 math.prime - Prime numbers .................................................................. 895
12.38 math.simplex - Simplex solver ............................................................... 897
12.39 os.windows - Windows support ............................................................. 898
  12.39.1 Windows dialogs .............................................................................. 898
12.39.2 Windows console API ........................................... 898
12.40  parser.peg - PEG parser combinators ............................ 902
  12.40.1 Walkthrough .................................................. 902
  12.40.2 Parser drivers ................................................ 907
  12.40.3 What is a PEG parser, really? ............................... 909
  12.40.4 Primitive parser builders ................................... 910
  12.40.5 Ropes .......................................................... 912
  12.40.6 Choice, backtrack and assertion combinators .............. 912
  12.40.7 Sequencing combinators ..................................... 913
  12.40.8 Repetition combinators ...................................... 915
  12.40.9 Miscellaneous combinators ................................. 916
  12.40.10 Performance ............................................... 917
12.41  rfc.822 - RFC822 message parsing ................................ 917
12.42  rfc.base64 - Base64 encoding/decoding .......................... 921
12.43  rfc.cookie - HTTP cookie handling ................................ 924
12.44  rfc.ftp - FTP client ............................................ 925
12.45  rfc.hmac - HMAC keyed-hashing ................................ 928
12.46  rfc.http - HTTP client .......................................... 930
  12.46.1 Http client mid-level API .................................. 930
  12.46.2 Http client utilities ....................................... 933
  12.46.3 Secure http connection ..................................... 935
12.47  rfc.icmp - ICMP packets ........................................ 935
12.48  rfc.ip - IP packets ............................................. 936
12.49  rfc.json - JSON parsing and construction ...................... 937
12.50  rfc.md5 - MD5 message digest .................................. 939
12.51  rfc.mime - MIME message handling ............................. 939
12.52  rfc.quoted-printable - Quoted-printable encoding/decoding 945
12.53  rfc.sha - SHA message digest ................................ 946
12.54  rfc.tls - Transport layer security ................................ 946
12.55  rfc.uri - URI parsing and construction ........................ 949
12.56  rfc.uuid - UUID .................................................. 954
12.57  rfc.zlib - zlib compression library ......................... 955
12.58  slib - SLIB interface ........................................... 959
12.59  sxml.ssax - Functional XML parser ............................. 959
  12.59.1 SSAX data types ............................................. 960
  12.59.2 SSAX low-level parsing code ................................ 962
  12.59.3 SSAX higher-level parsers and scanners ................... 966
  12.59.4 SSAX Highest-level parsers - XML to SXML ................ 967
12.60  sxml.xpath - SXML query language ................................ 970
  12.60.1 SXPath basic converters and applicators ................... 970
  12.60.2 SXPath query language ..................................... 974
  12.60.3 SXPath extension ........................................... 976
12.61  sxml.tools - Manipulating SXML structure ...................... 978
  12.61.1 SXML predicates ............................................ 978
  12.61.2 SXML accessors ............................................. 979
  12.61.3 SXML modifiers ............................................. 981
  12.61.4 SXPath auxiliary utilities .................................. 982
  12.61.5 SXML to markup conversion .................................. 982
12.62  sxml.serializer - Serializing XML and HTML from SXML ........ 983
  12.62.1 Simple SXML serializing .................................... 983
  12.62.2 Custom SXML serializing .................................... 984
12.63  text.console - Text terminal control .......................... 985
12.64  text.csv - CSV tables .......................................... 988
12.65  text.diff - Calculate difference of text streams .......... 991
12.66 text.edn - EDN parsing and construction ................................................. 994
12.67 text.external-editor - Running external editor ........................................... 997
12.68 text.gap-buffer - Gap buffer ..................................................................... 998
12.69 text.gettext - Localized messages ............................................................. 1000
12.70 text.html-lite - Simple HTML document construction .................................. 1002
12.71 text.info - Accessing info document ............................................................ 1004
12.72 text.multicolumn - Multicolumn formatting ................................................ 1006
12.73 text.pager - Display with pager .................................................................. 1006
12.74 text.parse - Parsing input stream .................................................................. 1007
12.75 text.progress - Showing progress on text terminals ..................................... 1008
12.76 text.segmented-match - Segmented string matching .................................... 1010
12.77 text.sql - SQL parsing and construction ..................................................... 1011
12.78 text.template - Simple template expander ................................................... 1012
12.79 text.tr - Transliterate characters ................................................................ 1013
12.80 text.tree - Lazy text construction ................................................................ 1014
12.81 util.combinations - Combination library ..................................................... 1015
12.82 util.digest - Message digester framework .................................................. 1017
12.83 util.dominator - Calculate dominator tree ................................................... 1018
12.84 util.isomorph - Determine isomorphism ...................................................... 1020
12.85 util.lcs - The longest common subsequence ................................................ 1020
12.86 util.levenshtein - Levenshtein edit distance ................................................ 1023
12.87 util.match - Pattern matching ...................................................................... 1024
12.88 util.record - SLIB-compatible record type .................................................... 1029
12.89 util.relation - Relation framework ............................................................... 1031
12.90 util.stream - Stream library ....................................................................... 1033
  12.90.1 Stream primitives .................................................................................. 1033
  12.90.2 Stream constructors .............................................................................. 1034
  12.90.3 Stream binding ..................................................................................... 1037
  12.90.4 Stream consumers ................................................................................ 1038
  12.90.5 Stream operations ............................................................................... 1038
12.91 util.temporal-relation - Temporal relation .................................................. 1043
12.92 util.toposort - Topological sort ................................................................... 1045
12.93 util.unification - Unification ...................................................................... 1046
12.94 www.cgi - CGI utility ............................................................................... 1047
12.95 www.cgi.test - CGI testing ....................................................................... 1052
12.96 www.css - CSS parsing and construction ................................................... 1053

Appendix A  C to Scheme mapping .......................................................... 1056

Appendix B  Function and Syntax Index ................................................... 1067

Appendix C  Module Index .............................................................. 1120

Appendix D  Lexical syntax index .......................................................... 1123

Appendix E  Class Index ................................................................. 1124

Appendix F  Variable Index ............................................................. 1127
1 Introduction

This is a users’ guide and reference manual of the Gauche Scheme. Here I tried to describe the implementation precisely, sometimes referring to background design choices.

The target readers are those who already know Scheme and want to write useful programs in Gauche. For those who are new to Scheme, it’ll be easier to start from some kind of tutorial. I’m planning to write one.

This manual only deals with Scheme side of things. Gauche has another face, a C interface. Details of it will be discussed in a separate document. See gauche-dev.texi in the source distribution for the work-in-progress of such document. Those who want to use Gauche as an embedded language, or want to write an extension, need that volume.

For the Scheme side, I tried to make this manual self-contained for the reader’s convenience, i.e. as far as you want to look up Gauche’s features you don’t need to refer to other documents. For example, description of functions defined in the standard documents are included in this manual, instead of saying “see the standard document”. However, this document is not a verbatim copy of the standard documents; sometimes I omit detailed discussions for brevity. I put pointers to the original documents, so please consult them if you need to refer to the standards.

If you’re reading this document off-line, you may find the most recent version on the web: https://practical-scheme.net/gauche/.

1.1 Overview of Gauche

Gauche is a Scheme script engine; it reads Scheme programs, compiles it on-the-fly and executes it on a virtual machine. Gauche conforms the language standard "Revised^7 Report on the Algorithmic Language Scheme" (https://bitbucket.org/cowan/r7rs/raw/tip/rnrs/r7rs.pdf), and supports various common libraries defined in SRFIs (https://srfi.schemers.org).

The goal of Gauche is to provide a handy tool for programmers and system administrators to handle daily works conveniently and efficiently in the production environment.

There are lots of Scheme implementations available, and each of them has its design emphasis and weaknesses. Gauche is designed with emphasis on the following criteria.

Quick startup

One of the situation Gauche is aiming at is in the production environment, where you write ten-lines throw-away script that may invoked very frequently. This includes CGI scripts as well. Gauche provides frequently used common features as a part of rich built-in functions or precompiled Scheme libraries that can be loaded very quickly.

Fully utilizing multi-core

Gauche supports native threads on most platforms. The internals are fully aware of preemptive/concurrent threads (that is, no “giant global lock”), so that you can utilize multiple cores on your machine.

Multibyte strings

We can no longer live happily in ASCII-only or 1-byte-per-character world. The practical language implementations are required to handle multibyte (wide) characters. Gauche supports multibyte strings natively, providing robust and consistent support than ad hoc library-level implementation. See Section 2.2 [Multibyte strings], page 13, for details.

Integrated object system

A powerful CLOS-like object system with MetaObject protocol (mostly compatible with STklos and Guile) is provided.
System interface

Although Scheme abstracts lots of details of the machine, sometimes you have to bypass these high-level layers and go down to the basement to make things work. Gauche has built-in support of most of POSIX.1 system calls. Other modules, such as networking module, usually provide both high-level abstract interface and low-level interface close to system calls.

Enhanced I/O

No real application can be written without dealing with I/O. Scheme neatly abstracts I/O as a port, but defines least operations on it. Gauche uses a port object as a unified abstraction, providing utility functions to operate on the underlying I/O system. See Section 6.21 [Input and output], page 259, for the basic I/O support.

Extended language

Gauche is not just an implementation of Scheme; it has some language-level enhancements. For example, lazy sequences allows you to have lazy data structures that behaves as if they’re ordinary lists (except that they’re realized lazily). It is different from library-level lazy structure implementation such as streams (SRFI-41), in a sense that you can use any list-processing procedures on lazy sequences. It enables programs to use lazy algorithms more liberally.

1.2 Notations

1.2.1 Entry format

In this manual, each entry is represented like this:

```
foo arg1 arg2
   [spec]{module} Description of foo . . .
```

Category denotes the category of the entry foo. The following categories will appear in this manual:

Function A Scheme function.
Special Form A special form (in the R7RS term, “syntax”).
Macro A macro.
Module A module
Class A class.
Generic Function A generic function
Method A method
Reader Syntax A lexical syntax that is interpreted by the reader.
Parameter A parameter, an object to realize dynamically-bound variables. See Section 6.16 [Parameters and dynamic states], page 232, for the details.
Dynamic state A procedure that follows a protocol to manage the state by dynamic scope. See Section 6.16 [Parameters and dynamic states], page 232, for the details.
Generic application In Gauche, you can “apply” a non-procedure object to arguments as if it is a procedure (see Section 6.15.6 [Applicable objects], page 225, for the details). This entry explains the behavior of an object applied to arguments.
Subprocess argument This appears in do-process and run-process to explain their keyword argument (see Section 9.27.1 [Running subprocess], page 488)
EC Qualifier  This is for SRFI-42 Eager Comprehension qualifiers.
(see Section 11.11 [Eager comprehensions], page 711).

For functions, special forms and macros, the entry may be followed by one or more arguments.
In the argument list, the following notations may appear:

arg ...  Indicates zero or more arguments.

:optional x y z
:optional (x x-default) (y y-default) z
Indicates it may take up to three optional arguments. The second form specifies
default values to x and y. This is Gauche’s enhancement to Scheme; see Section 4.3
[Making procedures], page 47, for the definition of complete argument list syntax.

:key x y z
:key (x x-default) (y y-default) z
Indicates it may take keyword arguments x, y and z. The second form shows the
default values for x and y. This is also Gauche’s enhancement to Scheme; see
Section 4.3 [Making procedures], page 47, for the definition of complete argument
list syntax.

:rest args
Indicates it may take rest arguments. This is also Gauche’s enhancement to Scheme;
see Section 4.3 [Making procedures], page 47, for the definition of complete argument
list syntax.

Following the entry line, we may indicate the specification the entry comes from, and/or the
module the entry is provided when it’s not built-in.

The specification is shown in brackets. You’ll see the following variations.

[R7RS], [R7RS library]
It is defined in R7RS. If the entry is about a procedure, a syntax or a macro, library
also shown to indicate the name is exported from the scheme.library module
(or the (scheme library) library, in R7RS terms).

[R7RS+], [R7RS+ library]
It is defined in R7RS, but extended by Gauche, e.g. accepting more optional argu-
ments or different type of arguments. The description contains how it is extended
from R7RS. When you’re writing a portable program, you need to be careful not to
use Gauche-specific features.

[R6RS], [R6RS+], [R5RS], [R5RS+]
It is defined in R6RS or R5RS. The plus sign means it has extended by Gauche.
Since R7RS is mostly upward-compatible to R5RS, and has a lot in common with
R6RS, we mark an entry as R5RS or R6RS only if it is not a part of R7RS.

[SRFI-n], [SRFI-n+]
The entry works as specified in SRFI-n. If it is marked as "[SRFI-n+]", the entry
has additional functionality.

[POSIX]  The API of the entry reflects the API specified in POSIX.

The module is shown in curly-braces. If the module isn’t shown, it is built-in for Gauche.
(Note: When you’re writing R7RS code, Gauche built-ins are available through (gauche base)
module, see Section 9.2 [Importing gauche built-ins], page 373).

Some entries may be available from more than one modules through re-exporting or module
inheritance. We only list the primary module in that case.

Here’s an actual entry for an example:

-- Function: utf8->string u8vector :optional start end


[R7RS base] \{gauche.unicode\} Converts a sequence of utf8 octets in UVECTOR to a string. Optional START and/or END argument(s) will limit the range of the input.

This shows the function utf8->string is specified by R7RS, in (scheme base) library. Gauche originally provides it from gauche.unicode module. You can import the function from either one, but in general, it’s good to use (import (scheme base)) when writing R7RS code, and (use gauche.unicode) when writing Gauche code. See Section 10.1 [R7RS integration], page 580, for the details of differences in writing in R7RS and Gauche.

1.2.2 Names and namespaces

Since R6RS, you can split toplevel definitions of Scheme programs into multiple namespaces. In the standards such namespaces are called libraries. Gauche predates R6RS and has been calling them modules, and we use the latter throughout this manual.

(Note: RnRS libraries are more abstract concept than Gauche’s modules; RnRS defines libraries in a way that they can be implemented in various ways, and it just happens that Gauche realises the library semantics using modules. When you write a portable R7RS library, be aware not to rely on Gauche-specific module semantics. Especially, RnRS libraries are more static than Gauche modules; you cannot add definitions to exiting libraries within RnRS, for example.)

Sometimes the same name is used for multiple definitions in different modules. If we need to distinguish those names, we prefix the name with the module name and a hash sign. For example, gauche#lambda means lambda defined in gauche module. This does not mean you can write gauche#lambda in the source code, though: This notation is just for explanation.
2 Concepts

In this chapter I describe a few Gauche’s design concepts that help you to understand how Gauche works.

2.1 Standard conformance

Gauche conforms “Revised^7 Report of Algorithmic Language Scheme,” (R7RS) including optional syntax and procedures. We cover R7RS small language (see Section 10.2 [R7RS small language], page 584), as well as part of R7RS large libraries (see Section 10.3 [R7RS large], page 593).

- Gauche has a special kind of symbols, called keywords. They’re symbols with its name beginning with a colon (e.g. :key), but behaves as if it is automatically bound to itself. See Section 6.8 [Keywords], page 157, for the details. Keywords are used extensively when passing so-called keyword arguments (see Section 4.3 [Making procedures], page 47).
- Continuations created in a certain situation (specifically, inside a Scheme code that is called from external C routine) have limited extent (See Section 6.15.7 [Continuations], page 226, for details).
- Full numeric tower (integer, rational, real and complex numbers) are supported, but rationals are only exact, and complex numbers are always inexact.

Note that, since Gauche predates R7RS, most existing Gauche source code doesn’t follow the R7RS program/library structure. Gauche can read both traditional Gauche modules/scripts and R7RS programs/libraries seamlessly. See Chapter 10 [Library modules - R7RS standard libraries], page 580, for the details of how R7RS is integrated into Gauche.

Gauche also supports the following SRFIs (Scheme Request for Implementation).

SRFI-0, Feature-based conditional expansion construct.

This has become a part of R7RS small. Gauche supports this as Built-in. See Section 4.12 [Feature conditional], page 75.

SRFI-1, List library (R7RS lists)

This has become a part of R7RS large. See Section 10.3.1 [R7RS lists], page 593. (Some of SRFI-1 procedures are built-in).

SRFI-2, AND-LET*: an AND with local bindings, a guarded LET* special form.

Supported natively. See Section 4.6 [Binding constructs], page 59.

SRFI-4, Homogeneous numeric vector datatypes.

The module gauche.uvector provides a superset of srFI.4 procedures, including arithmetic operations and generic interface on the SRFI-4 vectors. See Section 6.13.2 [Uniform vectors], page 198.

SRFI-5, A compatible let form with signatures and rest arguments

Supported by the module srFI.5. See Section 11.4 [A compatible let form with signatures and rest arguments], page 691.

SRFI-6, Basic String Ports.

This has become a part of R7RS small. Gauche supports this as built-in. See Section 6.21.5 [String ports], page 268.

SRFI-7, Feature-based program configuration language

Supported as an autoloaded macro. See Section 11.5 [Feature-based program configuration language], page 691.

SRFI-8, receive: Binding to multiple values.

Syntax receive is built-in. See Section 4.6 [Binding constructs], page 59.
SRFI-9, Defining record types.
   Supported by the module `gauche.record`. See Section 9.28 [Record types], page 502.

SRFI-10, Sharp-comma external form.
   Built-in. See Section 6.21.7.3 [Read-time constructor], page 273.

SRFI-11, Syntax for receiving multiple values.
   This has become a part of R7RS small. Gauche supports it as built-in. See Section 4.6 [Binding constructs], page 59.

SRFI-13, String library
   Supported by the module `srfi.13`. See Section 11.6 [String library], page 692.
   (Some of SRFI-13 procedures are built-in).

SRFI-14, Character-set library
   This has become a part of R7RS large. Character-set object and a few procedures are built-in. See Section 6.10 [Character sets], page 165. Complete set of SRFI-14 is supported by the module `scheme.charset`. See Section 10.3.6 [R7RS character sets], page 614.

SRFI-16, Syntax for procedures of variable arity (case-lambda)
   This has become a part of R7RS small. Built-in. See Section 4.3 [Making procedures], page 47.

SRFI-17, Generalized set!
   Built-in. See Section 4.4 [Assignments], page 52.

SRFI-18, Multithreading support
   Some SRFI-18 features are built-in, and the rest is in `gauche.threads` module. See Section 9.35 [Threads], page 530.

SRFI-19, Time Data Types and Procedures.
   Time data type is Gauche built-in (see Section 6.24.9 [Time], page 316). Complete set of SRFI-19 is supported by the module `srfi.19`. See Section 11.7 [Time data types and procedures], page 701.

SRFI-22, Running Scheme scripts on Unix
   Supported. See Section 3.3 [Writing Scheme scripts], page 31.

SRFI-23, Error reporting mechanism.
   This has become a part of R7RS small. Built-in. See Section 6.19.2 [Signaling exceptions], page 248.

SRFI-25, Multi-dimensional array primitives.
   Supported by the module `gauche.array`, which defines superset of SRFI-25. See Section 9.1 [Arrays], page 366.

SRFI-26, Notation for specializing parameters without currying.
   Built-in. See Section 4.3 [Making procedures], page 47.

SRFI-27, Sources of Random Bits.
   Supported by the module `srfi.27`. See Section 11.8 [Sources of random bits], page 706.

SRFI-28, Basic format strings.
   Gauche’s built-in `format` procedure is a superset of SRFI-28 `format`. See Section 6.21.8 [Output], page 275.

SRFI-29, Localization
   Supported by the module `srfi.29`. See Section 11.9 [Localization], page 707.
Chapter 2: Concepts

SRFI-30, Nested multi-line comments.
This has become a part of R7RS small. Supported by the native reader. See Section 4.1 [Lexical structure], page 43.

See Section 4.6 [Binding constructs], page 59.

SRFI-34, Exception Handling for Programs
This has become a part of R7RS small. Built-in. See Section 6.19 [Exceptions], page 245.

SRFI-35, Conditions
Built-in. See Section 6.19.4 [Conditions], page 253.

SRFI-36, I/O Conditions
Partly supported. See Section 6.19.4 [Conditions], page 253.

SRFI-37, args-fold: a program argument processor
Supported by the module srfi.37. See Section 11.10 [A program argument processor], page 709.

SRFI-38, External Representation for Data With Shared Structure
Built-in. See Section 6.21.7.1 [Reading data], page 270, and Section 6.21.8 [Output], page 275.

SRFI-39, Parameter objects
This has become a part of R7RS small. Built-in (see Section 6.16.1 [Parameters], page 233).

SRFI-40, A Library of Streams
Supported by the module util.stream. See Section 12.90 [Stream library], page 1033.

SRFI-41, Streams
This has become a part of R7RS large. See Section 10.3.14 [R7RS stream], page 635. Most of stream procedures are also in util.stream (see Section 12.90 [Stream library], page 1033).

SRFI-42, Eager comprehensions
Supported by the module srfi.42. See Section 11.11 [Eager comprehensions], page 711.

SRFI-43, Vector library
Supported by the module srfi.43. See Section 11.12 [Vector library (Legacy)], page 717. Note that this srfi is superseded by R7RS scheme.vector library (formerly known as srfi.133). See Section 10.3.2 [R7RS vectors], page 597.

SRFI-45, Primitives for Expressing Iterative Lazy Algorithms
Built-in. See Section 6.18 [Lazy evaluation], page 239.

SRFI-46, Basic Syntax-rules Extensions
This has become a part of R7RS small. Built-in. See Section 5.2 [Hygienic macros], page 90.

SRFI-55, require-extension
Supported as an autoloaded macro. See Section 11.13 [Requiring extensions], page 718.

SRFI-60, Integers as bits
Most procedures are built-in: See Section 10.3.22 [R7RS bitwise operations], page 664. The complete support is in srfi.60 module: See Section 11.14 [Integers as bits], page 718.
SRFI-61, A more general cond clause
Supported natively. See Section 4.5 [Conditionals], page 55.

SRFI-62, S-expression comments
This has become a part of R7RS small. Supported by the native reader. See Section 4.1 [Lexical structure], page 43.

SRFI-64, A Scheme API for test suites
Supported by the module srfi.64. See Section 11.15 [A Scheme API for test suites], page 720.

SRFI-66, Octet vectors
Supported by the module srfi.66 (see Section 11.16 [Octet vectors], page 720). This is mostly a subset of gauche.uvector, but has one slight difference.

SRFI-69, Basic hash tables
Supported by the module srfi.69 (see Section 11.17 [Basic hash tables], page 721). Note that this srfi is superseded by R7RS scheme.hash-table library (formerly known as srfi.125). See Section 10.3.7 [R7RS hash tables], page 618.

SRFI-74, Octet-addressed binary blocks
Supported by the module srfi.74 (see Section 11.18 [Octet-addressed binary blocks], page 723).

SRFI-78, Lightweight testing
Supported by the module srfi.78. It can work with gauche.test. See Section 11.19 [Lightweight testing], page 725.

SRFI-87, => in case clauses
This has become a part of R7RS small. Supported natively. See Section 4.5 [Conditionals], page 55.

SRFI-95, Sorting and merging
Supported natively. See Section 6.23 [Sorting and merging], page 291.

SRFI-96, SLIB Prerequisites
This srfi is not exactly a library, but rather a description of features the platform should provide to support SLIB. In order to load this module, SLIB must be already installed. See Section 12.58 [SLIB], page 959, for the details.

SRFI-98, An interface to access environment variables
Supported by the module srfi.98. See Section 11.20 [Accessing environment variables], page 727.

SRFI-99, ERR5RS Records
Supported by the module gauche.record. See Section 9.28 [Record types], page 502.

SRFI-101, Purely functional random-access pairs and lists
This has become a part of R7RS large. Supported by the module scheme.rlist (see Section 10.3.9 [R7RS random-access lists], page 622).

SRFI-106, Basic socket interface
Supported by the module srfi.106. See Section 11.22 [Basic socket interface], page 727.

SRFI-111, Boxes
This has become a part of R7RS large as scheme.box. Gauche has it as built-in. See Section 6.17 [Boxes], page 238.
SRFI-112, Environment inquiry
Supported by the module srfi.112. See Section 11.23 [Portable runtime environment inquiry], page 730.

SRFI-113, Sets and Bags
This has become a part of R7RS large. Supported by the module scheme.set. See Section 10.3.5 [R7RS sets], page 606.

SRFI-114, Comparators
Some of the features are built-in (see Section 6.2.4 [Basic comparators], page 116). Full srfi spec is supported by the module srfi.114 (see Section 11.24 [Comparators], page 731).

SRFI-115, Scheme Regular Expressions
This has become a part of R7RS large. Supported by the module scheme.regex. See Section 10.3.19 [R7RS regular expressions], page 640.

SRFI-116, Immutable List Library
This has become a part of R7RS large. Immutable pairs are supported natively (see Section 6.6.2 [Mutable and immutable pairs], page 141). Full set of APIs are available in the module scheme.ilist (see Section 10.3.8 [R7RS immutable lists], page 621).

SRFI-117, Queues based on lists.
This has become a part of R7RS large. Supported by the module scheme.list-queue, which is implemented on top of data.queue. (see Section 10.3.16 [R7RS list queues], page 636)

SRFI-118, Simple adjustable-size strings
Supported by the module srfi.118. (see Section 11.25 [Simple adjustable-size strings], page 736)

SRFI-120, Timer APIs
Supported by the module srfi.120 (see Section 11.26 [Timer APIs], page 736). It is a wrapper of control.scheduler (see Section 12.11 [Scheduler], page 824).

SRFI-121, Generators
This has become a part of R7RS large. Gauche’s gauche.generator is superset of srfi-121 (see Section 9.12 [Generators], page 436).

SRFI-124, Ephemerons
This has become a part of R7RS large. Supported by scheme.ephemeron. Note: Current Gauche’s implementation isn’t optimal. See Section 10.3.17 [R7RS ephemeros], page 639.

SRFI-125, Intermediate hash tables
This has become a part of R7RS large. Supported by scheme.hash-table (see Section 10.3.7 [R7RS hash tables], page 618). Note that Gauche’s native interface provides the same functionalities, but under slightly different names for the backward compatibility. See Section 6.14.1 [Hashtables], page 206.

SRFI-127, Lazy sequences
This has become a part of R7RS large. Supported by scheme.lseq (see Section 10.3.13 [R7RS lazy sequences], page 633).

SRFI-128, Comparators (reduced)
This has become a part of R7RS large. Built-in. See Section 6.2.4 [Basic comparators], page 116, for the details.
SRFI-129, Titlecase procedures

The procedures `char-title-case?` and `char-titlecase` are built-in, and `string-titlecase` is in `gauche.unicode`. For the compatibility, you can `(use srfi.129)` or `(import (srfi 129))` to get these three procedures.

SRFI-130, Cursor-based string library

String cursors are supported natively (see Section 6.11.5 [String cursors], page 174). Most of built-in and srfi-13 string procedures accept cursors in addition to indexes. The module srfi-130 provides several procedures that has the same name as srfi-13 but returns a string cursor instead of an index (see Section 11.28 [Cursor-based string library], page 738).

SRFI-131, ERR5RS Record Syntax (reduced)

This srfi is a pure subset of srfi-99, and `gauche.record`'s `define-record-type` covers it. See Section 9.28 [Record types], page 502.

SRFI-132, Sort libraries

This has become a part of R7RS large. Supported by the module `scheme.sort`. See Section 10.3.4 [R7RS sort], page 602.

SRFI-133, Vector library (R7RS-compatible)

This has become a part of R7RS large. Supported by the module `scheme.vector`. See Section 10.3.2 [R7RS vectors], page 597.

SRFI-134, Immutable Deques

This has become a part of R7RS large. The module `data.ideque` is compatible to srfi-134. See Section 12.16 [Immutable deques], page 833.

SRFI-135, Immutable Texts

This has become a part of R7RS large. In Gauche, the text type is not disjoint from the string type. Instead, a text is simply an immutable and indexed string. See Section 6.11.6 [String indexing], page 176, for the detail of indexed string. The API is described in Section 10.3.11 [R7RS immutable texts], page 627.

SRFI-141, Integer division

This has become a part of R7RS large. Supported by the module `scheme.division`. See Section 10.3.21 [R7RS integer division], page 663.

SRFI-143, Finxums

This has become a part of R7RS large. Supported by the module `scheme.fixnum`. See Section 10.3.23 [R7RS fixnum], page 668.

SRFI-144, Flonums

This has become a part of R7RS large. Supported by the module `scheme.flonum`. See Section 10.3.24 [R7RS flonum], page 670.

SRFI-145, Assumptions

Built-in. See Section 4.5 [Conditionals], page 55.

SRFI-146, Mappings

This has become a part of R7RS large. Supported by the module `scheme.mapping`. See Section 10.3.20 [R7RS mappings], page 652.

SRFI-149, Basic syntax-rules template extensions

The built-in `syntax-rules` support srfi.149.

SRFI-151, Bitwise operations

Supported by the module `srfi.151` (see Section 10.3.22 [R7RS bitwise operations], page 664). Note that many equivalent procedures are provided built-in (see Section 6.3.6 [Basic bitwise operations], page 136).
SRFI-152, String library (reduced)
Supported by the module srfi.152 (see Section 11.29 [String library (reduced)], page 740).

SRFI-154, First-class dynamic extents
Supported by the module srfi.154. (see Section 11.30 [First-class dynamic extents], page 742).

SRFI-158, Generators and accumulators
This has become a part of R7RS large. Supported by the module scheme.generator (see Section 10.3.12 [R7RS generators], page 631). Note that most of generator procedures are supported by gauche.generator (see Section 9.12 [Generators], page 436).

SRFI-159, Combinator Formatting
This has become a part of R7RS large. See Section 10.3.26 [R7RS combinator formatting], page 681.

SRFI-160, Homogeneous numeric vector libraries
This has become a part of R7RS large, supported by the module scheme.vector. where @ is one of base, u8, s8, u16, s16, u32, s32, u64, s64, f32, f64, c64, or c128 (see Section 10.3.3 [R7RS uniform vectors], page 602).

SRFI-162, Comparators sublibrary
Supported by the module srfi.162. See Section 11.32 [Comparator sublibrary], page 744.

SRFI-169, Underscores in numbers
Supported by the built-in reader. See Section 4.1 [Lexical structure], page 43.

SRFI-170, POSIX API
Supported by the module srfi.170. See Section 11.33 [POSIX API], page 744.

SRFI-172, Two safer subsets of R7RS
Supported by the modules srfi.172 and srfi.172.functional. See Section 11.34 [Two safer subsets of R7RS], page 750.

SRFI-173, Hooks
Supported by the module srfi.173 (see Section 11.35 [Hooks (srfi)], page 751), which is a thin layer on top of gauche.hook (see Section 9.13 [Hooks], page 447)

SRFI-174, POSIX Timespecs
Supported by the module srfi.174 (see Section 11.36 [POSIX timespecs], page 752). In Gauche, the timespec type is the same as built-in <time> object, which is also the same as srfi-19 time.

SRFI-175, ASCII character library
Supported by the module srfi.175 (see Section 11.37 [ASCII character library], page 753).

SRFI-176, Version flag
Supported as a command-line flag of gosh. The version-alist procedure is built-in.

SRFI-178, Bitvector library
The basic support is built-in (see Section 6.13.3 [Bitvectors], page 203). Complete support is in the module srfi.178 (see Section 11.38 [Bitvector library], page 756.

SRFI-180, JSON
Supported by the module srfi.180 (see Section 11.39 [JSON], page 757). Note that Gauche also has rfc.json, and srfi.180 is implemented on top of it.
SRFI-181, Custom ports
Supported by the module `srfi.181` (see Section 11.40 [Custom ports], page 760). Gauche has an original custom port mechanism (see Section 9.40 [Virtual ports], page 572), and this srfi is built on top of it.

SRFI-185, Linear adjustable-length strings
Supported by the module `srfi.185`. See Section 11.41 [Linear adjustable-length strings], page 764.

SRFI-189, Maybe and Either: optional container types
Supported by the module `srfi.189`. See Section 11.42 [Maybe and Either optional container types], page 765.

SRFI-192, Port positioning
Gauche's port already has positioning mechanism, so main procedures are built-in (see Section 6.21.3 [Common port operations], page 260). A few additional procedures are provided by the module `srfi.192` (see Section 11.43 [Port positioning], page 773).

SRFI-193, Command line
Two procedures, `command-line` and `script-file`, are built-in. Other APIs are provided by the module `srfi.193` (see Section 11.44 [Command line], page 773).

SRFI-195, Multiple-value boxes
Built-in. See Section 6.17 [Boxes], page 238.

SRFI-196, Range objects
Supported by the module `srfi.196`. Also the `data.range` module is the superset of this srfi (see Section 12.21 [Range], page 845).

SRFI-197, Pipeline operators
Supported by the module `srfi.197` (see Section 11.46 [Pipeline operators], page 774).

SRFI-207, String-notated bytevectors
Bytestring literals are supported by the core. Bytestring procedures are available in the module `srfi.207` (see Section 11.47 [String-notated bytevectors], page 776).

SRFI-209, Enums and enum sets
Supported by the module `srfi.209` (see Section 11.48 [ Enums and enum sets], page 779).

SRFI-210, Procedures and syntax for multiple values
Supported by the module `srfi.210` (see Section 11.49 [Procedures and syntax for multiple values], page 784).

SRFI-216, SICP Prerequisites
Supported by the module `srfi.216` (see Section 11.50 [SICP prerequisites], page 786).

SRFI-217, Integer sets
Supported by the module `srfi.217` (see Section 11.51 [Integer sets], page 787).

SRFI-219, Define higher-order lambda
Gauche's built-in `define` (both R7RS-compatible one and extended one) supports this feature. If you import srfi-219 explicitly, it imports `null#define`. See Section 11.52 [Define higher-order lambda], page 792, for the details.

SRFI-221, Generator/accumulator sub-library
Supported by the module `srfi.221` (see Section 11.53 [Generator/accumulator sub-library], page 792).
SRFI-222, Compound objects
   Supported by the module srfi.222 (see Section 11.54 [Compound objects], page 794).

SRFI-226, Control features
   Partially supported as built-in and gauche.threads module. See Section 11.55 [Control features], page 796, for the details.

SRFI-227, Optional arguments
   Supported by the module srfi.227 (see Section 11.56 [Optional arguments], page 800).

SRFI-228, Composing comparators
   Supported by the module srfi.228 (see Section 11.57 [Composing comparators], page 801).

SRFI-229, Tagged procedures
   Supported by the module srfi.229 (see Section 11.58 [Tagged procedures], page 802).

SRFI-232, Flexible curried procedures
   Supported by the module srfi.232 (see Section 11.59 [Flexible curried procedures], page 803).

SRFI-235, Combinators (SRFI)
   Supported by the module srfi.235 (see Section 11.60 [Combinators (SRFI)], page 804).

SRFI-236, Evaluating expressions in an unspecified order
   Built-in. See Section 4.7 [Grouping], page 62.

SRFI-239, Destructuring lists
   Supported by the module srfi.239 (see Section 11.61 [Destructuring lists], page 807).

SRFI-244, Multiple-value definitions
   Built-in. See Section 4.10 [Definitions], page 68.

2.2 Multibyte strings

Traditionally, a string is considered as a simple array of bytes. Programmers tend to imagine a string as a simple array of characters (though a character may occupy more than one byte). It’s not the case in Gauche.

Gauche supports multibyte string natively, which means characters are represented by variable number of bytes in a string. Gauche retains semantic compatibility of Scheme string, so such details can be hidden, but it’ll be helpful if you know a few points.

A string object keeps a type tag and a pointer to the storage of the string body. The storage of the body is managed in a sort of “copy-on-write” way—if you take substring, e.g. using directly by substring or using regular expression matcher, or even if you copy a string by copy-string, the underlying storage is shared (the “anchor” of the string is different, so the copied string is not eq? to the original string). The actual string is copied only if you destructively modify it.

Consequently the algorithm like pre-allocating a string by make-string and filling it with string-set! becomes extremely inefficient in Gauche. Don’t do it. (It doesn’t work with multibyte strings anyway). Sequential access of string is much more efficient using string ports (see Section 6.21.5 [String ports], page 268).
String search primitives such as *string-scan* (see Section 6.11.9 [String utilities], page 178) and regular expression matcher (see Section 6.12 [Regular expressions], page 183) can return a matched string directly, without using index access at all.

You can choose *internal* encoding scheme at the time of compiling Gauche. At runtime, a procedure *gauche-character-encoding* can be used to query the internal encoding. At compile time, you can use a feature identifier to check the internal encoding. (see Section 3.5 [Platform-dependent features], page 34.) Currently, the following internal encodings are supported.

- **utf-8** UTF-8 encoding of Unicode. This is the default. The feature identifier *gauche.ces.utf8* indicates Gauche is compiled with this internal encoding.
- **euc-jp** EUC-JP encoding of ASCII, JIS X 0201 kana, JIS X 0212 and JIS X 0213:2000 Japanese character set. The feature identifier *gauche.ces.eucjp* indicates Gauche is compiled with this internal encoding.
- **sjis** Shift-JIS encoding of JIS X 0201 kana and JIS X 0213:2000 Japanese character set. For source-code compatibility, the character code between 0 and 0x7f is mapped to ASCII. The feature identifier *gauche.ces.sjis* indicates Gauche is compiled with this internal encoding.
- **none** 8-bit fixed-length character encoding, with the code between 0 and 0x7f matches ASCII. It’s up to the application to interpret the string with certain character encodings. The feature identifier *gauche.ces.none* indicates Gauche is compiled with this internal encoding.

Conversions from other encoding scheme is provided as a special port. See Section 9.5 [Character code conversion], page 399, for details.

The way to specify the encoding of source programs will be explained in the next section.

## 2.3 Multibyte scripts

You can use characters other than *us-ascii* not only in literal strings and characters, but in comments, symbol names, literal regular expressions, and so on.

By default, Gauche assumes a Scheme program is written in its internal character encoding. It is fine as far as you’re writing scripts to use your own environment, but it becomes a problem if somebody else tries to use your script and finds out you’re using different character encoding than his/hers.

So, if Gauche finds a comment something like the following within the first two lines of the program source, it assumes the rest of the source code is written in `<encoding-name>`, and does the appropriate character encoding conversion to read the source code:

```
;;; coding: <encoding-name>
```

More precisely, a comment in either first or second line that matches a regular expression `#/coding[:=]s*(\w-)+#` is recognized, and the first submatch is taken as an encoding name. If there are multiple matches, only the first one is effective. The first two lines must not contain characters other than us-ascii in order for this mechanism to work.

The following example tells Gauche that the script is written in EUC-JP encoding. Note that the string " -*- " around the coding would be recognized by Emacs to select the buffer’s encoding appropriately.

```
#!/usr/bin/gosh
;;; -*- coding: euc-jp -*-

... script written in euc-jp ...
```
Internally, the handling of this *magic comment* is done by a special type of port. See Section 6.21.6 [Coding-aware ports], page 270, for the details. See also Section 6.22.1 [Loading Scheme file], page 286, for how to disable this feature.

### 2.4 Case-sensitivity

Historically, most Lisp-family languages are case-insensitive for symbols. Scheme departed from this tradition since R6RS, and the symbols are read in case-sensitive way. (Note that symbols have been case-sensitive internally even in R5RS Scheme; case-insensitivity is about readers.)

Gauche reads and writes symbols in case-sensitive manner by default, too. However, to support legacy code, you can set the reader to case-insensitive mode, in the following ways:

- **Use `#!fold-case` reader directive**
  
  When Gauche sees a token `#!fold-case` during reading a program, the reader switches to case-insensitive mode. A token `#!no-fold-case` has an opposite effect—to make the reader case-sensitive. These tokens affect the port from which they are read, and are in effect until EOF or another instance of these tokens are read. See Section 4.1 [Lexical structure], page 43, for more details on `#!` syntax. This is the way defined in R6RS and R7RS.

- **Use `-fcase-fold` command-line argument**
  
  Alternatively, you can give a command-line argument `-fcase-fold` to the `gosh` command (see Section 3.1 [Invoking Gosh], page 19). In this mode, the reader folds uppercase characters in symbols to lowercase ones. If a symbol name contains uppercase characters, it is written out using `|escape` (see Section 6.7 [Symbols], page 155).

### 2.5 Integrated object system

Gauche has a STklos-style object system, similar to CLOS. If you have used some kind of object oriented (OO) languages, you'll find it easy to understand the basic usage:

```scheme
;; Defines a class point, that has x and y coordinate
(define-class point ()
  ((x :init-value 0)
   (y :init-value 0)))

(define-method move ((p point) dx dy)
  (inc! (slot-ref p 'x) dx)
  (inc! (slot-ref p 'y) dy))

(define-method write-object ((p point) port)
  (format port "[point ~a ~a]"
          (slot-ref p 'x)
          (slot-ref p 'y)))
```

However, if you are familiar with mainstream OO languages but new to CLOS-style object system, Gauche's object system may look strange when you look deeper into it. Here I describe several characteristics of Gauche object system quickly. See Chapter 7 [Object system], page 329, for details.

*Everything is an object (if you care)*

You have seen this tagline for the other languages. And yes, in Gauche, everything is an object in the sense that you can query its class, and get various meta information
of the object at run time. You can also define a new method on any class, including built-in ones.

Note that, however, in CLOS-like paradigm it doesn’t really matter whether everything is an object or not, because of the following characteristics:

**Method is dispatched by all of its arguments.**

Unlike other object-oriented languages such as C++, Objective-C, Python, Ruby, etc., in which a method always belong to a single class, a Gauche method doesn’t belong to a specific class.

For example, suppose you define a numeric vector class `<num-vector>` and a numeric matrix class `<num-matrix>`. You can define a method `product` with all possible combinations of those type of arguments:

- `(product <num-vector> <num-matrix>)`
- `(product <num-matrix> <num-vector>)`
- `(product <num-vector> <num-vector>)`
- `(product <num-matrix> <num-matrix>)`
- `(product <number> <num-vector>)`
- `(product <number> <num-matrix>)`
- `(product <number> <number>)`

Each method belongs to neither `<num-vector>` class nor `<num-matrix>` class.

Since a method is not owned by a class, you can always define your own method on the existing class (except a few cases that the system prohibits altering pre-defined methods). The above example already shows it; you can make `product` method work on the built-in class `<number>`. That is why I said it doesn’t make much sense to discuss whether everything is object or not in CLOS-style object system.

To step into the details a bit, the methods are belong to a **generic function**, which is responsible for dispatching appropriate methods.

**Class is also an instance.**

By default, a class is also an instance of class `<class>`, and a generic function is an instance of class `<generic>`. You can subclass `<class>` to customize how a class is initialized or how its slots are accessed. You can subclass `<generic>` to customize how the applicable methods are selected, which order those methods are called, etc. The mechanism is called **metaobject protocol**. Metaobject protocol allows you to extend the language by the language itself.

To find examples, see the files `lib/gauche/singleton.scm` and `lib/gauche/mop/validator.scm` included in the distribution. You can also read `lib/gauche/mop/object.scm`, which actually defines how a class is defined in Gauche. For more details about metaobject protocol, see Gregor Kiczales, Jim Des Rivieres, Daniel Bobrow, The Art of Metaobject Protocol, The MIT Press.

**Class doesn’t create namespace**

In the mainstream OO language, a class often creates its own namespace. This isn’t the case in CLOS-style object system. In Gauche, a namespace is managed by the module system which is orthogonal to the object system.

### 2.6 Module system

Gauche has a simple module system that allows modularized development of large software.

A higher level interface is simple enough from the user’s point of view. It works like this: When you want to use the features provided by module `foo`, you just need to say `(use foo)` in
your code. This form is a macro and interpreted at compile time. Usually it loads the files that defines foo’s features, and imports the external APIs into the calling module.

The use mechanism is built on top of two independent lower mechanisms, namespace separation and file loading mechanism. Those two lower mechanisms can be used separately, although it is much more convenient when used together.

The use mechanism is not transitive; that is, if a module B uses a module A, and a module C uses the module B, C doesn’t see the bindings in A. It is because B and A is not in the is-a relationship. Suppose the module A implements a low-level functionality and the module B implements a high-level abstraction; if C is using B, what C wants to see is just a high-level abstraction, and doesn’t concern how B implements such functionality. If C wants to access low-level stuff, C has to use A explicitly.

There is another type of relationship, though. You might want to take an exiting module A, and add some interface to it and provide the resulting module B as an extension of A. In such a case, B is-a A, and it’d be natural that the module that uses B can also see A’s bindings. In Gauche, it is called module inheritance and realized by extend form.

The following sections in this manual describes modules in details.

- Section 3.7 [Writing Gauche modules], page 38, explains the convention of writing modules.
- Section 4.13 [Modules], page 78, describes special forms and macros to define and to use modules, along the built-in functions to introspect module internals.

2.7 Compilation

By default, Gauche reads toplevel Scheme forms one at a time, compile it immediately to intermediate form and execute it on the VM. As long as you use Gauche interactively, it looks like an interpreter. (There’s an experimental ahead-of-time compiler as well. See HOWTO-precompile.txt if you want to give a try.)

The fact that we have separate compilation/execution phase, even interleaved, may lead a subtle surprise if you think Gauche as an interpreter. Here’s a few points to keep in mind:

load is done at run time.

load is a procedure in Gauche, therefore evaluated at run time. If the loaded program defines a macro, which is available for the compiler after the toplevel form containing load is evaluated. So, suppose foo.scm defines a macro foo, and you use the macro like this:

;;; in “foo.scm”
(define-syntax foo
  (syntax-rules () ((arg) (quote arg))))

;;; in your program
(begin (load "foo") (foo (1 2 3)))
⇒ error, bad procedure: ‘1’

(load "foo")
(foo (1 2 3)) ⇒ ’(1 2 3)

The (begin (load ...)) form fails, because the compiler doesn’t know foo is a special form at the compilation time and compiles (1 2 3) as if it is a normal procedure call. The latter example works, however, since the execution of the toplevel form (load "foo") is done before (foo (1 2 3)) is compiled.

To avoid this kind of subtleties, use require or use to load a program fragments. Those are recognized by the compiler.
**require is done at compile time**

On the other hand, since `require` and `use` is recognized by the compiler, the specified file is loaded even if the form is in the conditional expression. If you really need to load a file on certain condition, use `load` or do dispatch in macro (e.g. `cond-expand` form (see Section 4.12 [Feature conditional], page 75). )
3 Programming in Gauche

3.1 Invoking Gosh

Gauche can be used either as an independent Scheme scripting engine or as an embedded Scheme library. An interactive interpreter which comes with Gauche distribution is a program named gosh.

```
gosh [options] [scheme-file arg . . .]  [Program]
```

Gauche’s interpreter. Without `scheme-file`, `gosh` works interactively, i.e. it reads a Scheme expression from the standard input, evaluates it, and prints the result, and repeat that until it reads EOF or is terminated.

If `gosh` is invoked without `scheme-file`, but the input is not a terminal, it enters read-eval-print loop but not writes out a prompt while waiting input form. This is useful when you pipe Scheme program into `gosh`. You can force this behavior or suppress this behavior by `-b` and `-i` options.

If `scheme-file` is specified, `gosh` runs it as a Scheme program and exit. See Section 3.3 [Writing Scheme scripts], page 31, for details.

Command-line options

The following command line options are recognized by `gosh`. The first command line argument which doesn’t begin with ‘−’ is recognized as the script file. If you want to specify a file that begins with a minus sign, use a dummy option ‘−−’.

- `−I` `path` [Command Option]
  Prepends `path` to the load path list. You can specify this option more than once to add multiple paths.

- `−A` `path` [Command Option]
  Appends `path` to the tail of the load path list. You can specify this option more than once to add multiple paths.

- `−q` [Command Option]
  Makes `gosh` not to load the default initialization file.

- `−V` [Command Option]
  Prints the `gosh` version and exits.

- `−v` `version` [Command Option]
  If `version` is not the running `gosh`’s version, execute the specified version of `gosh` instead if it is installed. This is useful when you want to invoke specific version of Gauche. Note that `version` must be 0.9.6 or later.

- `−u` `module` [Command Option]
  Use `module`. Before starting execution of `scheme-file` or entering the read-eval-print loop, the specified module is used, i.e. it is loaded and imported (See Section 4.13.3 [Defining and selecting modules], page 81, for details of `use`). You can specify this option more than once to use multiple modules.

- `−l` `file` [Command Option]
  Load `file` before starting execution of `scheme-file` or entering the read-eval-print loop. The file is loaded in the same way as `load` (see Section 6.22.1 [Loading Scheme file], page 286). You can specify this option more than once to load multiple files.
-L file
   Load file like -l, but if file does not exist, this silently ignores it instead of reporting an error.
   This option can also be specified multiple times.

-e scheme-expression
   Evaluate scheme-expression before starting execution of scheme-file or entering the read-eval-print loop.
   Evaluation is done in the interaction-environment (see Section 6.20 [Eval and repl], page 258). You can specify this option more than once to evaluate multiple expressions.

-E scheme-expression
   Same as -e, except the scheme-expression is read as if it is surrounded by parenthesis. For example:
   ```
   % gosh -umath.const -E"print (sin (* pi/180 15))" -Eexit
   0.25881904510252074
   ```

-b
   Batch. Does not print prompts even if the input is a terminal.

-i
   Interactive. Print prompts even if the input is not a terminal.

-m module
   When a script file is given, this option makes the module named module in which the main
   procedure is looked for, instead of the user module. See Section 3.3 [Writing Scheme scripts],
   page 31, for the details of executing scripts.
   If the named module doesn’t exist after loading the script, an error is signaled.
   This is useful to write a Scheme module that can also be executed as a script.

-f compiler-option
   This option controls compiler and runtime behavior. For now we have following options available:
   case-fold  Ignore case for symbols.
   include-verbose
      Reports whenever a file is included. Useful to check precisely which files are included in what order.
   load-verbose
      Reports whenever a file is loaded. Useful to check precisely which files are loaded in what order.
   no-inline
      Prohibits the compiler from inlining procedures and constants. Equivalent to
      no-inline-globals, no-inline-locals, no-inline-constants and no-inline-setters combined.
   no-inline-constants
      Prohibits the compiler from inlining constants.
   no-inline-globals
      Prohibits the compiler from inlining global procedures.
   no-inline-locals
      Prohibits the compiler from inlining local procedures.
   no-inline-setters
      Prohibits the compiler from inlining setters.
no-lambda-lifting-pass
Prohibits the compiler from running lambda-lifting pass.

no-post-inline-pass
Prohibits the compiler from running post-inline optimization pass.

no-source-info
Don’t keep source information for debugging. Consumes less memory.

safe-string-cursors
String cursors used on wrong strings will raise an error. This may cause performance problems because all cursors will be allocated on heap. See Section 6.11.5 [String cursors], page 174.

test
Adds ".../src" and ".../lib" to the load path before loading initialization file. This is useful when you want to test the compiled gosh REPL without installing it.

warn-legacy-syntax
Warns if the reader sees legacy hex-escape syntax in string literals. See Section 6.21.7.2 [Reader lexical mode], page 272. See Section 2.4 [Case-sensitivity], page 15.

-p profiler-option
[Command Option]
Turn on the profiler. The following profiler-option is recognized:

time
Records and reports time spent on function calls and number of times each function is called.

load
Records and reports time spent on loading each modules. Useful to tune start-up time of the scripts. (Results are in elapsed time).

See Section 3.6.1 [Using profiler], page 36, for the details of the profiler.

-r standard-revision
[Command Option]
Start gosh with an environment of the specified revision of Scheme standard. Currently only 7 is supported as standar-revision.

By default, gosh starts with user module, which inherits gauche module. That means you can use whole Gauche core procedures by default without explicitly declaring it.

Proper R7RS code always begins with either define-library or R7RS-style import form, and Gauche recognizes it and automatically switch to R7RS environments so that R7RS scripts and libraries can be executed by Gauche without special options. However, users who are learning R7RS Scheme may be confused when the initial environment doesn’t look like R7RS.

By giving -r7 option, gosh starts with r7rs.user module that extends the r7rs module, which defines two R7RS forms, import and define-library.

If you invoke gosh into an interactive REPL mode with -r7 option, all standard R7RS-small libraries (except (scheme r5rs)) are already imported for your convenience.

See Chapter 10 [Library modules - R7RS standard libraries], page 580, for the details on how Gauche supports R7RS.

(Note: The -r7 option doesn’t change reader lexical mode (see Section 6.21.7.2 [Reader lexical mode], page 272) to strict-r7. That’s because using strict-r7 mode by default prevents many Gauche code from being loaded.)

--
[Command Option]
When gosh sees this option, it stops processing the options and takes next command line argument as a script file. It is useful in case if you have a script file that begins with a minus sign, although it is not generally recommended.
The options -I, -A, -l, -u, -e and -E are processes in the order of appearance. For example, adding a load path by -I affects the -l and -u option after it but not before it.

Environment variables

The following environment variables are recognized:

**GAUCHE\_AVAILABLE\_PROCESSORS**
You can get the number of system’s processors by `sys-available-processors` (see Section 6.24.3 [Environment inquiry], page 295); libraries/programs may use this info to optimize number of parallel threads. But you might change that, for testing and benchmarking—e.g. a program automatically uses 8 threads if there are 8 cores, but you might want to run it with 1, 2, 4 threads as well to see the effect of parallelization. This environment variable overrides the return value of `sys-available-processors`.

**GAUCHE\_CHECK\_UNDEFINED\_TEST**
Warn if `#<undef>` is used in the test expression of branch.

In boolean context, `#<undef>` counts true. It is also often the case that a procedure returns `#<undef>` when the return value doesn’t matter, and you shouldn’t rely on the value that is supposed not to matter—the procedure may change the return value in future (which should be ok, since the value shouldn’t have mattered), which can cause unintentional and hard-to-track bugs. See Section 6.5 [Undefined values], page 140, for the details.

We strongly recommend users to turn on this warning. In future, we plan to make this default.

**GAUCHE\_DYNLOAD\_PATH**
You can specify additional load paths for dynamically loaded objects by this environment variable, delimiting the paths by ‘:’. The paths are appended before the system default load paths.

See Section 6.22.2 [Loading dynamic library], page 287, for the details of how Gauche finds dynamically loadable objects.

**GAUCHE\_EDITOR**
This is used by `ed` procedure in `gauche.interactive` module. See Section 9.14 [Interactive session], page 448, for the details.

**GAUCHE\_LEGACY\_DEFINE**
Make the behavior of toplevel `define` the same as 0.9.8 and before. It allows certain legacy programs that aren’t valid R7RS. See Section 4.10.1 [Into the Scheme-Verse], page 72, for the details.

**GAUCHE\_LOAD\_PATH**
You can specify additional load paths by this environment variable, delimiting the paths by ‘:’. The paths are appended before the system default load paths.
See Section 6.22.1 [Loading Scheme file], page 286, for the details of how Gauche finds files to load.

**GAUCHE_MUTABLE_LITERALS**

[Environment variable]

Allow literal lists and vectors to be mutated. Such code isn’t a valid Scheme program and causes an error, but Gauche didn’t enforce the restriction on 0.9.9 and before, so some legacy code may accidentally mutate literals. Set this environment variable to run such old programs. See Section 4.2 [Literals], page 46, for the details.

**GAUCHE_NO_READ_EDIT**

[Environment variable]

Disable line-editor on REPL prompt, even the terminal is capable. You can also turn it off with `-fno-read-edit` command-line option, or `,edit off` toplevel commands during REPL session. See Section 3.2 [Interactive development], page 24, for the details of line editing.

**GAUCHE_QUASIRENAME_MODE**

[Environment variable]

This affects `quasirename` behavior, to keep the backward compatibility with 0.9.7 and before. See Section 5.2.2 [Explicit-renaming macro transformer], page 93, for the details.

**GAUCHE_REPL_NO_PPRINT**

[Environment variable]

This is used by `gauche.interactive` module to suppress pretty-printing on REPL prompt. See Section 3.2 [Interactive development], page 24, for the details.

**GAUCHE_SUPPRESS_WARNING**

[Environment variable]

Suppress system warnings (WARNING: ...). Not generally recommended; use only if you absolutely need to.

**GAUCHE_TEST_RECORD_FILE**

[Environment variable]

This is used by `gauche.test` module (see Section 9.34 [Unit testing], page 521). If defined, names a file the test processes keep the total statistics.

**GAUCHE_TEST_REPORT_ERROR**

[Environment variable]

This is used by `gauche.test` module (see Section 9.34 [Unit testing], page 521). If defined, reports stack trace to stderr when the test thunk raises an error (even when it is expected). Useful for diagnosis of unexpected errors.

**TMP**

[Environment variable]

**TMPDIR**

[Environment variable]

**TEMP**

[Environment variable]

**USERPROFILE**

These may affect the return value of `sys-tmpdir`. Different environment variables may be used on different platforms. See Section 6.24.4.3 [Pathnames], page 300, for the details.

### Windows-specific executable

On Windows-native platforms (mingw), two interpreter executables are installed. `gosh.exe` is compiled as a Windows console application and works just like ordinary `gosh`; that is, it primarily uses standard i/o for communication. Another executable, `gosh-noconsole.exe`, is compiled as a Windows no-console (GUI) application. It is not attached to a console when it is started. Its standard input is connected to the `NUL` device. Its standard output and standard error output are special ports which open a new console when something is written to them for the first time. (NB: This magic only works for output via Scheme ports; direct output from low-level C libraries will be discarded.)

The main purpose of `gosh-noconsole.exe` is for Windows scripting. If a Scheme script were associated to `gosh.exe` and invoked from Explorer, it would always open a new console window, which is extremely annoying. If you associate Scheme scripts to `gosh-noconsole.exe` instead, you can avoid console from popping up.
If you’re using the official Windows installer, Scheme scripts (*.scm) have already associated to gosh-noconsole.exe and you can invoke them by double-clicking on Explorer. Check out some examples under C:\Program Files\Gauche\examples.

### 3.2 Interactive development

When gosh is invoked without any script files, it goes into interactive read-eval-print loop (REPL).

To exit the interpreter, type EOF (usually Control-D in Unix terminals) or evaluate (exit).

In the interactive session, gosh loads and imports gauche.interactive module (see Section 9.14 [Interactive session], page 448) into user module, for the convenience. Also, if there’s a file .gaucherc under the user’s home directory. You may put settings there that would help interactive debugging. (As of Gauche release 0.7.3, .gaucherc is no longer loaded when gosh is run in script mode.)

Note that .gaucherc is always loaded in the user module, even if gosh is invoked with -r7 option. The file itself is a Gauche-specific feature, so you don’t need to consider portability in it.

I recommend you to run gosh inside Emacs, for it has rich features useful to interact with internal Scheme process. Put the following line to your .emacs file:

```
(setq scheme-program-name "gosh -i")
```

And you can run gosh by M-x run-scheme.

If you run gosh in the terminal with capability of cursor control, a basic line-editing feature is available in the REPL session. See Section 3.2.2 [Input editing], page 29, for the details.

If you want to use multibyte characters in the interaction, make sure your terminal’s settings is in sync with gosh’s internal character encodings.

#### 3.2.1 Working in REPL

When you enter REPL, Gauche prompts you to enter a Scheme expression:

```
gosh> (+ 1 2)
gosh> 3
```

The REPL session binds the last three results of evaluation in the global variables *1, *2 and *3. You can use the previous results via those history variables in subsequent expressions.

```
gosh> *1
3
```

If the Scheme expression yields multiple values (see Section 6.15.10 [Multiple values], page 230), they are printed one by one.

```
gosh> (min&max 1 -1 8 3)
-1
3
```

```
gosh> 
```
The history variable *1, *2 and *3 only binds the first value. A list of all values are bound to *1+, *2+ and *3+.

```
gosh> *1
-1
```
```
gosh> *2+
(-1 8)
```

(Note that, when you evaluate *1 in the above example, the history is shifted—so you need to use *2+ to refer to the result of (min&max 1 -1 8 3).)

The *history procedure shows the value of history variables:

```
gosh> (*history)
*1: (-1 8)
*2: -1
*3: -1
```

As a special case, if an evaluation yields zero values, history isn’t updated. The *history procedure returns no values, so merely looking at the history won’t change the history itself.

```
gosh> (values)
```
```
gosh> (*history)
*1: (-1 8)
*2: -1
*3: -1
```

Finally, a global variable *e is bound to the last uncaught error condition object.

```
gosh> (filter odd? '(1 2 x 4 5))
*** ERROR: integer required, but got x
Stack Trace:
```
```
0 (eval expr env)
   At line 173 of "/usr/share/gauche-0.9/0.9.3.3/lib/gauche/interactive.scm"
```
```
gosh> *e
#<error "integer required, but got x">
```

(The error stack trace may differ depending on your installation.)

In REPL prompt, you can also enter special top-level commands for common tasks. Top-level commands are not Scheme expressions, not even S-expressions. They work like traditional line-oriented shell commands instead.

Top-level commands are prefixed by comma to be distinguished from ordinary Scheme expressions. To see what commands are available, just type ,help and return.

```
gosh> ,help
You’re in REPL (read-eval-print-loop) of Gauche shell.
Type a Scheme expression to evaluate.
A word preceded with comma has special meaning. Type ,help <cmd> to see the detailed help for <cmd>.
Commands can be abbreviated as far as it is not ambiguous.
```
```
,apropos|a Show the names of global bindings that match the regexp.
,cd Change the current directory.
```
,describe|d Describe the object.
,help|h Show the help message of the command.
,history>Show REPL history.
,info|doc Show info document for an entry of NAME, or search entries by REGEXP.
,load|l Load the specified file.
,print-all|pa
Print previous result (*1) without abbreviation.
,print-mode|pm
View/set print-mode of REPL.
,pwd Print working directory.
,reload|r Reload the specified module, using gauche.reload.
,sh Run command via shell.
,source Show source code of the procedure if it’s available.
,use|u Use the specified module. Same as (use module option ...).

To see the help of each individual commands, give the command name (without comma) to
the help command:

gosh> ,help d
Usage: d|describe [object]
Describe the object.
Without arguments, describe the last REPL result.

The ,d (or ,describe) top-level command describes the given Scheme object or the last
result if no object is given. Let’s try some:

gosh> (sys-stat "/home")
#<<sys-stat> 0x2d6adc0>
gosh> ,d
#<<sys-stat> 0x2d6adc0> is an instance of class <sys-stat>
slots:
  type : directory
  perm : 493
  mode : 16877
  ino : 2
  dev : 2081
  rdev : 0
  nlink : 9
  uid : 0
  gid : 0
  size : 208
  atime : 1459468837
  mtime : 1401239524
  ctime : 1401239524

In the above example, first we evaluated (sys-stat "/home"), which returns <sys-stat>
object. The subsequent ,d top-level command describes the returned <sys-stat> object.

The description depends on the type of objects. Some types of objects shows extra information.
If you describe an exact integer, it shows alternative interpretations of the number:

gosh> ,d 1401239524
1401239524 is an instance of class <integer>
  (#x538537e4, ~ 1.3Gi, 2014-05-28T01:12:04Z as unix-time)
gosh> ,d 48
48 is an instance of class <integer>
  (#x30, #\0 as char, 1970-01-01T00:00:48Z as unix-time)
If you describe a symbol, its known bindings is shown.

```gosh
,g,d 'filter
filter is an instance of class <symbol>
Known bindings for variable filter:
In module 'gauche':
  #<closure (filter pred lis)> 
In module 'gauche.collection':
  #<generic filter (2)>
```

If you describe a procedure, and its source code location is known, that is also shown (see the Defined at... line):

```gosh
,g,d string-interpolate
#<closure (string-interpolate str :optional (legacy? #f))> is an 
instance of class <procedure>
Defined at "../lib/gauche/interpolate.scm":64
slots:
  required : 1
  optional : #t
  optcount : 1
  locked : #f
  currying : #f
  constant : #f
  info : (string-interpolate str :optional (legacy? #f))
  setter : #f
```

Let's see a couple of other top-level commands. The ,info command shows the manual entry of the given procedure, variable, syntax, module or a class. (The text is searched from the installed info document of Gauche. If you get an error, check if the info document is properly installed.)

```gosh
,g,info append
-- Function: append list ...
[R7RS] Returns a list consisting of the elements of the first LIST 
followed by the elements of the other lists. The resulting list is 
always newly allocated, except that it shares structure with the 
last list argument. The last argument may actually be any object; 
an improper list results if the last argument is not a proper list.
```

```gosh
,g,info srfi.19
-- Module: srfi.19
This SRFI defines various representations of time and date, and 
conversion methods among them.

On Gauche, time object is supported natively by '<time>' class 
(*note Time::). Date object is supported by '<date>' class 
described below.
```

```gosh
,g,info <list>
-- Builtin Class: <list>
An abstract class represents lists. A parent class of '<null>' and 
'<pair>'. Inherits '<sequence>'.

Note that a circular list is also an instance of the '<list>' 
class, while 'list?' returns false on the circular lists and dotted
lists.

( use srfi.1)
(list? (circular-list 1 2)) => #f
(is-a? (circular-list 1 2) <list>) => #t

You can also give a regexp pattern to ,info command (see Section 6.12 [Regular expressions], page 183). It shows the entries in the document that match the pattern.

gosh> ,info #/^

The ,a command (or ,apropos) shows the global identifiers matches the given name or regexp:

gosh> ,a filter
filter (gauche)
filter! (gauche)
filter$ (gauche)
filter-map (gauche)

Note: The apropos command looks for symbols from the current process—that is, it only shows names that have been loaded and imported. But it also mean it can show any name as far as it exists in the current process, regardless of whether it’s a documented API or an internal entry.

On the other hand, the info command searches info document, regardless of the named entity has loaded into the current process or not. It doesn’t show undocumented APIs.

You can think that apropos is an introspection tool, while info is a document browsing tool.

When the result of evaluation is a huge nested structure, it may take too long to display the result. Gauche actually set a limit of length and depth in displaying structures, so you might occasionally see the very long or deep list is truncated, with ... to show there are more items, or # to show a subtree is omitted (Try evaluating (make-list 100) on REPL).

You can type ,pa (or ,print-all) toplevel REPL command to fully redisplay the previous result without omission.

By default, REPL prints out the result using pretty print:

gosh> ,u sxml.ssax

gosh> (call-with-input-file "src/Info.plist" (cut ssax:xml->sxml <> '(())))
(*TOP*)
(*PI* xml "version=\"1.0\" encoding=\"UTF-8\"")
If you want to turn off pretty printing for some reason, type `pm pretty #f` (or `print-mode pretty #f`) on the toplevel prompt, or start `gosh` with the environment variable `GAUCHE_REPL_NO_PPRINT` set.

Type `pm default` to make print mode back to default. For more details, type `help pm`.

Note: If you invoke `gosh` with `-q` option, which tells `gosh` not to load the initialization files, you still get a REPL prompt but no fancy features such as history variables are available. Those convenience features are implemented in `gauche.interactive` module, which isn’t loaded with `-q` option.

### 3.2.2 Input editing

When you run `gosh` in a terminal capable of cursor control, you can edit input expressions. If input editing mode is on, the REPL prompt ends with `$`, such as `gosh$`, instead of `gosh>`.

(NB: Currently Gauche only supports terminals with vt100-like escape sequence, or Windows console. If the terminal type isn’t recognized as one of them, it falls back to non-editing mode. You can tell which mode it is from the prompt.)

The input editing feature is still under development. If you stumbled with a serious bug, you can turn it off by setting an environment variable `GAUCHE_NO_READ_EDIT`, or giving `-fno-read-edit` option to `gosh`, or type `,edit off` on REPL.

The key binding is similar to Emacs. Eventually we’ll provide customization feature. Before going into details, here’s a few quick useful tips.

- If the screen is garbled somehow, type `C-1` (control+l) to clear and redisplay.
- If you want to turn off editing during REPL session, use `,edit off` toplevel command.
- The input editor only sends a complete S-expression to the evaluator. If somehow you want to send the current input as-is (or, in case the editor has a bug and don’t allow you to send a complete S-expression), type `C-M-x` (control-meta-x) to force sending the current input to the evaluator.
- You can type `M-h h` to see a brief summary of editor features, `M-h b` to see the list of keymap, or `M-h k +` keystroke to see the help of the key. (`C-h` is the same as backspace).
- If you suspend `gosh`, then resume it by shell’s job control feature (e.g. `C-z`, then `fg`). type `ENTER` to regain editing screen.

**Cursor movement**

`C-f` (forward) and `C-b` (backward) moves the cursor forward and backward character-wise. `C-p` (previous) and `C-n` (next) moves character to previous or next line, if the input already has multiple lines, or moves to previous or next history.

`M-f` and `M-b` move the cursor forward and backward, word-wise.

`C-a` and `C-e` to move to the beginning and end of the line, `M-<` and `M->` to move to the beginning and end of the input.
Undo

C-_ is undo the edit. You can keep typing C-_ to undo the edits you’ve made. We follow the Emacs model of undo semantics, which allows “undoing undoes”. For the detailed algorithm, see the comment at the bottom of `lib/text/line-edit.scm` in the source tree.

Kill and yank

C-k removes characters from the cursor to the end of line. If the cursor is at the end of the line though, it removes the newline character (so that next line is combined to the current line).

M-d removes a word that contains the cursor, or a word immediately after the cursor if it is not on a word.

C-@ set a mark to the current cursor position. C-w removes characters between the cursor and the mark.

The characters removed by those commands are saved in the buffer called “kill-ring”. They can be recalled at the cursor position by C-y (yank). If you press M-y immediately followed by C-y, you can go back to the older killed characters.

Interpolation

Typing TAB in or immediately after a partially typed word completes it. If there are multiple candidates, it completes up to the common prefix, and typing TAB again shows the list of candidates.

What’s to be completed depends on the context. If it is at the toplevel command (a word after a comma on a prompt), it completes toplevel commands. If it is argument of `load` toplevel command, it completes pathnames. If it is argument of `use` or `reload` toplevel commands, it completes module names. In other contexts, it completes global bindings visible from the current module.

For symbols and module names, segmented completion is possible; e.g. c-w-c-c completes to `call-with-current-continuation`.

Currently the completion mechanism is hardcoded. We may add a way to customize it in future.

Finishing input

RET (or C-m) inserts a newline if the input isn’t a complete S-expression. If the input is already a complete S-expression, however, it sends the entire input to the evaluator, no matter where the cursor is. If you want to insert a newline in a complete S-expression, you can use C-j.

M-C-x sends the current input regardless that the input is a complete S-expression or not.

History

Input history is remembered and recalled by M-p (prev-history) and M-n (next-history). The cursor movement command C-p and C-n also moves to the previous or next history if it is pressed when the cursor is at the beginning or the end of the input lines.

The input interrupted by C-c isn’t remembered.

By default, the input is saved to a file `~/.gosh_history` when the REPL is terminated normally, and reloaded when the next REPL is invoked. The name of the history file can be changed by the environment variable GAUCHE_HISTORY_FILE. If the environment variable is defined to an empty string, however, the history won’t be saved.

Miscellaneous

C-g cancels the current multi-key sequences.

C-c cancels the current input.
C-t transpose characters at and before the cursor.
C-q reads the next keystroke and insert it into the input as is.
M-( inserts a pair of parentheses, and locate a cursor inside them.
C-1 clears the screen and redraws the current input buffer.

3.3 Writing Scheme scripts

When a Scheme program file is given to gosh, it makes the user module as the current module, binds a global variable *argv* to the list of the remaining command-line arguments, and then loads the Scheme program. If the first line of scheme-file begins with two character sequence “#!”, the entire line is ignored by gosh. This is useful to write a Scheme program that works as an executable script in unix-like systems.

Typical Gauche script has the first line like these

```plaintext
#!/usr/local/bin/gosh
or,
#!/usr/bin/env gosh
or,
#!/bin/sh
exec gosh -- $0 "$@"
```

The second and third form uses a “shell trampoline” technique so that the script works as far as gosh is in the PATH. The third form is useful when you want to pass extra arguments to gosh, for typically #!-magic of executable scripts has limitations for the number of arguments to pass the interpreter.

After the file is successfully loaded, gosh calls a procedure named ’main’ if it is defined in the user module. Main receives a single argument, a list of command line arguments. Its first element is the script name itself.

When main returns, and its value is an integer, gosh uses it for exit code of the program. Otherwise, gosh exits with exit code 70 (EX_SOFTWARE). This behavior is compatible with the SRFI-22.

If the main procedure is not defined, gosh exits after loading the script file.

Although you can still write the program main body as toplevel expressions, like shell scripts or Perl scripts, it is much convenient to use this ‘main’ convention, for you can load the script file interactively to debug.

Using -m command-line option, you can make gosh call main procedure defined in a module other than the user module. It is sometimes handy to write a Scheme module that can also be executed as a script.

For example, you write a Scheme module foo and within it, you define the main procedure. You don’t need to export it. If the file is loaded as a module, the main procedure doesn’t do anything. But if you specify -m foo option and give the file as a Scheme script to gosh, then the main procedure is invoked after loading the script. You can code tests or small example application in such an alternate main procedure.

Note on R7RS Scripts: If the script is written in R7RS Scheme (which can be distinguished by the first import declaration, see Section 10.1.2 [Three forms of import], page 582), it is read into r7rs.user module and its main isn’t called. You can give -mr7rs.main command-line argument to call the main function in R7RS script. Alternatively, as specified in SRFI-22, if the script interpreter’s basename is scheme-r7rs, we assume the script is R7RS SRFI-22 script and calls main in r7rs.user module rather than user module. We don’t install such an alias, but you can manually make symbolic link or just copy gosh binary as scheme-r7rs.
Although the argument of the `main` procedure is the standard way to receive the command-line arguments, there are a couple of other ways to access to the info. See Section 6.24.2 [Command-line arguments], page 294, for the details.

Now I show several simple examples below. First, this script works like `cat(1)`, without any command-line option processing and error handling.

```groff
#!/usr/bin/env gosh
(define (main args) ; entry point
  (if (null? (cdr args))
    (copy-port (current-input-port) (current-output-port))
    (for-each (lambda (file)
      (call-with-input-file file
        (lambda (in)
          (copy-port in (current-output-port))))))
    (cdr args)))
0)
```

The following script is a simple grep command.

```groff
#!/usr/bin/env gosh
(define (usage program-name)
  (format (current-error-port)
    "Usage: ~a regexp file ...
" program-name)
  (exit 2))

(define (grep rx port)
  (with-input-from-port port
    (lambda ()
      (port-for-each
        (lambda (line)
          (when (rxmatch rx line)
            (format #t "~a:~a: ~a
" (port-name port)
              (- (port-current-line port) 1)
              line))))
    read-line))))

(define (main args)
  (if (null? (cdr args))
    (usage (car args))
    (let ((rx (string->regexp (cadr args))))
      (if (null? (cddr args))
        (grep rx (current-input-port))
        (for-each (lambda (f)
          (call-with-input-file f
            (lambda (p) (grep rx p))))
          (cddr args))))))
0)
```

See also Section 9.25 [Parsing command-line options], page 480, for a convenient way to parse command-line options.
3.4 Debugging

Gauche doesn’t have much support for debugging yet. The idea of good debugging interfaces are welcome.

For now, the author uses the classic ’debug print stub’ technique when necessary. Gauche’s reader supports special syntaxes beginning with `#?`, to print the intermediate value.

The syntax `#?=expr` shows `expr` itself before evaluating it, and prints its result(s) after evaluation.

```
gosh> #?= (+ 2 3)
#?="(stdin)”:1: (+ 2 3)
#?- 5

5

gosh> #?= (begin (print "foo") (values ’a ’b ’c))
#?="(stdin)”:2: (begin (print "foo") (values ’a ’b ’c))
foo
#?- a
#?+ b
#?+ c
a
b
c
```

Note: If the debug stub is evaluated in a thread other than the primordial thread (see Section 9.35 [Threads], page 530), the output includes a number to distinguish which thread it is generated. In the following example, `#<thread ...>` and the prompt is the output of REPL in the primordial thread, but following `#?=[1]...` and `#?-[1]...` are the debug output from the thread created by `make-thread`. The number is for debugging only—they differ for each thread, but other than that there’s no meaning.

```
gosh> (use gauche.threads)
gosh> (thread-start! (make-thread (^[] #?= (+ 2 3))))
#<thread #f (1) runnable 0xf51400>
gosh> #?=[1]"(standard input)”:1: (+ 2 3)
#?- [1] 5
```

The syntax `#?,(proc arg ...)`, is specifically for procedure call; it prints the value of arguments right before calling proc, and prints the result(s) of call afterwards.

```
gosh> (define (fact n)
    (if (zero? n)
        1
        (* n #?,(fact (- n 1)))))
fact
#?,"(standard input)”:4: calling ‘fact’ with args:
#?,> 4
#?,"(standard input)”:4: calling ‘fact’ with args:
#?,> 3
#?,"(standard input)”:4: calling ‘fact’ with args:
#?,> 2
#?,"(standard input)”:4: calling ‘fact’ with args:
#?,> 1
#?,"(standard input)”:4: calling ‘fact’ with args:
#?,> 0
#?- 1
#?- 1
```
Internally, the syntax #?=x and #?,x are read as (debug-print x) and (debug-funcall x), respectively, and the macros debug-print and debug-funcall handles the actual printing. See Section 6.25.1 [Debugging aid], page 326, for more details.

The reasons of special syntax are: (1) It’s easy to insert the debug stub, for you don’t need to enclose the target expression by extra parenthesis, and (2) It’s easy to find and remove those stabs within the editor.

Simple debug stubs may produce too many output, and you may want to limit them when certain conditions are met. A reader syntax #??=test expr and #??,test procedure-call works much like #??=expr and #?,procedure-call, but only when test yields true:

```
(define (fib n)
  (if (< n 2)
    1
    #??,(= n 5) (+ (fib (- n 1)) (fib (- n 2)))))
```

```
gosh> (fib 7)
#?,calling ‘+’ with args:
#?,> 5
#?,> 3
#?- 8
#?,calling ‘+’ with args:
#?,> 5
#?,> 3
#?- 8
21
```

Internally, #??=test expr and #??,test procedure-call are expanded to macro calls (debug-print-conditionally test expr) and (debug-funcallt-conditionally test procedure-call).

### 3.5 Using platform-dependent features

Gauche tries to provide low-level APIs close to what the underlying system provides, but sometimes they vary among systems. For example, POSIX does not require symlink, so some systems may lack sys-symlink (see Section 6.24.4.2 [Directory manipulation], page 299). Quite a few unix-specific system functions are not available on Windows platform. To allow writing a portable program across those platforms, Gauche uses cond-expand (see Section 4.12 [Feature conditional], page 75) extensively. A set of extended feature-identifiers is provided to check availability of specific features. For example, on systems that has symlink, a feature identifier gauche.sys.symlink is defined. So you can write a code that can switch based on the availability of sys-symlink as follows:

```
(cond-expand
  (gauche.sys.symlink
    ... code that uses sys-symlink ...)
  (else
    ... alternative code ...)
)
```

If you’re familiar with system programming in C, you can think it equivalent to the following C idiom:
#if defined(HAVE_SYMLINK)
... code that uses symlink ...
#else
... alternative code ...
#endif

There are quite a few such feature identifiers; each identifier is explained in the manual entry of the procedures that depend on the feature. Here we list a few important ones:

gauche  This feature identifier is always defined. It is useful when you write Scheme code portable across multiple implementations.

gauche.os.windows  Defined on Windows native platform. Note that cygwin does not define this feature identifier (but see below).

gauche.os.cygwin  Defined on Cygwin.

gauche.sys.threads  Defined if Gauche is compiled with thread support. See Section 9.35 [Threads], page 530.

gauche.sys.pthreads
gauche.sys.wthreads  Defined to indicate the underlying thread implementation when Gauche has thread support. See Section 9.35 [Threads], page 530.

gauche.net.ipv6  Defined if Gauche is compiled with IPv6 support.

gauche.ces.utf8
gauche.ces.eucjp
gauche.ces.sjis
gauche.ces.none  Either one of these feature identifiers is defined, according to the compile-time option of Gauche’s internal character encoding. See Section 2.2 [Multibyte strings], page 13, for the details of the internal character encoding.

Because `cond-expand` is a macro, the body of clauses are expanded into toplevel if `cond-expand` itself is in toplevel. That means you can switch toplevel definitions:

```
(cond-expand
  (gauche.os.windows
   (define (get-current-user)
    ... get current username ...))
  (else
   (define (get-current-user)
    (sys-uid->user-name (sys-getuid)))))
```

Or even conditionally "use" the modules:

```
(cond-expand
  (gauche.os.windows
   (use "my-windows-compatibility-module")
  (else))
```

The traditional technique of testing a toplevel binding (using `global-variable-bound?`, see Section 4.13.6 [Module introspection], page 84) doesn’t work well in this case, since the `use` form takes effect at compile time. It is strongly recommended to use `cond-expand` whenever possible.
Currently the set of feature identifiers are fixed at the build time of Gauche, so it’s less flexible than C preprocessor conditionals. We have a plan to extend this feature to enable adding new feature identifiers; but such feature can complicate semantics when compilation and execution is interleaved, so we’re carefully assessing the effects now.

A couple of notes:

Feature identifiers are not variables. They can only be used within the feature-requirement part of cond-expand (see Section 4.12 [Feature conditional], page 75, for the complete definition of feature requirements).

By the definition of SRFI-0, cond-expand raises an error if no feature requirements are satisfied and there’s no else clause. A rule of thumb is to provide else clause always, even it does nothing (like the above example that has empty else clause).

3.6 Profiling and tuning

If you find your script isn’t running fast enough, there are several possibilities to improve its performance.

It is always a good idea to begin with finding which part of the code is consuming the execution time. Gauche has a couple of basic tools to help it. A built-in sampling profiler, explained in the following subsection, can show how much time is spent in each procedure, and how many times it is called. The gauche.time module (Section 9.36 [Measure timings], page 546) provides APIs to measure execution time of specific parts of the code.

Optimization is specialization—you look for the most common patterns of execution, and put a special path to support those patterns efficiently. Gauche itself is no exception, so there are some patterns Gauche can handle very efficiently, while some patterns it cannot. The next subsection, Section 3.6.2 [Performance tips], page 37, will give you some tips of how to adapt your code to fit the patterns Gauche can execute well.

3.6.1 Using profiler

As of 0.8.4, Gauche has a built-in profiler. It is still experimental quality and only be tested on Linux. It isn’t available for all platforms. It works only in single-thread applications for now.

To use the profiler non-interactively, give -ptime command-line option to gosh.

% gosh -ptime your-script.scm

After the execution of your-script.scm is completed, Gauche prints out the table of functions with its call count and its consumed time, sorted by the total consumed time.

Profiler statistics (total 1457 samples, 14.57 seconds)

<table>
<thead>
<tr>
<th>Name</th>
<th>num calls</th>
<th>time/call(ms)</th>
<th>total samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>combinations*</td>
<td>237351</td>
<td>0.0142</td>
<td>337( 23%)</td>
</tr>
<tr>
<td>(lset-difference #f)</td>
<td>1291837</td>
<td>0.0020</td>
<td>256( 17%)</td>
</tr>
<tr>
<td>(make-anchor make-anchor)</td>
<td>3950793</td>
<td>0.0005</td>
<td>198( 13%)</td>
</tr>
<tr>
<td>member</td>
<td>4627246</td>
<td>0.0004</td>
<td>190( 13%)</td>
</tr>
<tr>
<td>filter</td>
<td>273238</td>
<td>0.0030</td>
<td>81( 5%)</td>
</tr>
<tr>
<td>every</td>
<td>1315131</td>
<td>0.0004</td>
<td>59( 4%)</td>
</tr>
<tr>
<td>(lset-difference #f #f)</td>
<td>1281837</td>
<td>0.0004</td>
<td>54( 3%)</td>
</tr>
<tr>
<td>(make-entry make-entry)</td>
<td>730916</td>
<td>0.0005</td>
<td>40( 2%)</td>
</tr>
<tr>
<td>(clear? #f)</td>
<td>730884</td>
<td>0.0005</td>
<td>33( 2%)</td>
</tr>
<tr>
<td>(initialize #f)</td>
<td>599292</td>
<td>0.0005</td>
<td>32( 2%)</td>
</tr>
<tr>
<td>fold</td>
<td>237307</td>
<td>0.0013</td>
<td>30( 2%)</td>
</tr>
<tr>
<td>acons</td>
<td>806406</td>
<td>0.0004</td>
<td>29( 1%)</td>
</tr>
<tr>
<td>clear?</td>
<td>33294</td>
<td>0.0084</td>
<td>28( 1%)</td>
</tr>
<tr>
<td>(combinations* #f)</td>
<td>805504</td>
<td>0.0002</td>
<td>15( 1%)</td>
</tr>
<tr>
<td>(make-exit make-exit)</td>
<td>730884</td>
<td>0.0002</td>
<td>15( 1%)</td>
</tr>
<tr>
<td>lset-difference</td>
<td>237318</td>
<td>0.0006</td>
<td>15( 1%)</td>
</tr>
</tbody>
</table>
Note that the time profiler uses statistic sampling. Every 10ms the profiler interrupts the process and records the function that is executed then. Compared to the individual execution time per function call, which is the order of nanoseconds, this sampling rate is very sparse. However, if we run the program long enough, we can expect the distribution of samples per each function approximately reflects the distribution of time spent in each function.

Keep in mind that the number is only approximation; the number of sample counts for a function may easily vary if the program deals with different data sets. It should also be noted that, for now, GC time is included in the function in which GC is triggered. This sometimes causes a less important function to "float up" to near-top of the list. To know the general pattern, it is a good custom to run the program with several different data sets.

On the other hand, the call count is accurate since Gauche actually counts each call.

Because all functions are basically anonymous in Scheme, the 'name' field of the profiler result is only a hint. The functions bound at toplevel is generally printed with the global variable name it is bound at the first time. Internal functions are printed as a list of names, reflecting the nesting of functions. Methods are also printed as a list of the name and specializers.

The profiler has its own overhead; generally the total process time will increase 20-30%. If you want to turn on the profiler selectively, or you're running a non-stop server program and want to obtain the statistics without exiting the server, you can call the profiler API from your program; see Section 6.25.2 [Profiler API], page 327, for the details.

### 3.6.2 Performance tips

Don’t guess, just benchmark. It is the first rule of performance tuning. Especially for the higher-level languages like Scheme, what impacts on performance greatly depends on the implementation. Certain operations that are very cheap on an implementation may be costly on others. Gauche has such implementation-specific characteristics, and to know some of them would help to see what to look out in the benchmark results.

"80% of execution time is spent in 20% of the code" is another old saying. Don’t obscure your code by "potential" optimization that has little impact on the actual execution. We describe some tips below, but it doesn’t mean you need to watch them all the time. It is better to keep most of the code clean and easy to understand, and only do tricks on the innermost loop.

**Ports:** To satisfy the specification of SRFI-18 (Threading), every call to I/O primitives of Gauche locks the port. This overhead may be visible if the application does a lot of I/O with smaller units (e.g. every bytes). The primitives that deals with larger unit, such as read and read-uvector, are less problematic, since usually they just lock the port once per call and do all the low-level I/O without the lock overhead. (Note: this doesn’t mean those primitives guarantee to lock the port throughout the execution of the function; furthermore, the port locking feature is optimized for the case that port accesses rarely collide. If you know it is possible that more than one threads read from or write to the same port, it is your responsibility to use mutex explicitly to avoid the collision.)

If you find out the locking is indeed a bottleneck, there are couple of things you can consider: (1) Try using the larger-unit primitives, instead of calling the smaller-unit ones. (2) Use with-port-locking (see Section 6.21.2 [Port and threads], page 259) to lock the port in larger context.

**Strings:** Because of the multibyte strings, two operations are particularly heavy in Gauche: string mutation and indexed string access. It is a design choice; we encourage the programming
style that avoids those operations. When you sequentially access the string, string ports (see Section 6.21.5 [String ports], page 268) provide a cleaner and more efficient way. When you search and retrieve a substring, there are various higher-level primitives are provided (see Section 6.11.9 [String utilities], page 178, Section 6.12 [Regular expressions], page 183, and Section 11.6 [String library], page 692, for example). If you’re using strings to represent an octet sequence, use uniform vectors (see Section 6.13.2 [Uniform vectors], page 198) instead.

**Deep recursion:** Gauche’s VM uses a stack for efficient local frame allocation. If recursion goes very deep (depending on the code, but usually several hundreds to a thousand), the stack overflows and Gauche moves the content of the stack into the heap. This incurs some overhead. If you observe a performance degradation beyond a certain amount of data, check out this possibility.

**Generic functions:** Because of its dynamic nature, generic function calls are slower than procedure calls. Not only because of the runtime dispatch overhead, but also because Gauche’s compile-time optimizer can’t do much optimization for generic function calls. You don’t need to avoid generic functions because of performance reasons in general, but if you do find single function call consuming a large part of execution time and it is calling a generic function in its inner loop—then it may be worth to modify it.

**Redefining builtin functions:** Gauche inlines some builtin functions if they are not redefined. Although sometimes it is useful to redefine basic functions, you may want to limit the effect. For example, put redefined functions in a separate module and use the module in the code that absolutely needs those functions replaced.

**Closure creation:** When you create a closure, its closing environment is copied to the heap. This overhead is small, but it still may be visible when a closure is created within an innermost loop that is called millions of times. If you suspect this is a problem, try disassemble the function. Gauche’s compiler uses some basic techniques of closure analysis to avoid creating closures for typical cases, in which case you see the local function’s bodies are inlined. If you see a CLOSURE instruction, though, it means a closure is created.

This list isn’t complete, and may change when Gauche’s implementation is improved, so don’t take this as fixed features. We’ll adjust it occasionally.

### 3.7 Writing Gauche modules

Gauche’s libraries are organized by modules. Although Gauche can load any valid Scheme programs, there is a convention that Gauche’s libraries follow. When you write a chunk of Scheme code for Gauche, it is convenient to make it a module, so that it can be shared and/or reused.

Usually a module is contained in a file, but you can make a multi-file module. First I explain the structure of a single-file module. The following template is the convention used in Gauche’s libraries.

```scheme
;; Define the module interface
(define-module foo
  (use xxx)
  (use yyy)
  (export foo1 foo2 foo3)
)

;; Enter the module
(select-module foo)

... module body ...
```
This file must be saved as “foo.scm” in some directory in the *load-path*.

The `define-module` form creates a module `foo`. It also loads and imports some other modules by `use` macros, and declares which symbols the `foo` module exports, by `export` syntax. (See section Section 4.13.3 [Defining and selecting modules], page 81, for detailed specification of those syntaxes).

Those `use` forms or `export` forms are not required to appear in the `define-module` form, but it is a good convention to keep them in there at the head of the file so that it is visually recognizable which modules `foo` depends and which symbols it exports.

The second form, ‘select-module’, specifies the rest of the file is evaluated in the module `foo` you just defined. Again, this is just a convention; you can write entire module body inside `define-module`. However, I think it is error-prone, for the closing parenthesis can be easily forgotten or the automatic indentation mechanism of editor will be confused.

After `select-module` you can write whatever Scheme expression. It is evaluated in the selected module, `foo`. Only the bindings of the exported symbols will be directly accessible from outside.

So, that’s it. Other programs can use your module by just saying `(use foo)`. If you want to make your module available on your site, you can put it to the site library location, which can be obtained by

```scheme
(gauche-site-library-directory)
```

in gosh, or

```shell
gauge-config --sitelibdir
```

from shell.

If you feel like to conserve global module name space, you can organize modules hierarchically. Some Gauche libraries already does so. See Chapter 8 [Library modules - Overview], page 359, for examples. For example, `text.tr` module is implemented in “text/tr.scm” file. Note that the pathname separator ‘/’ in the file becomes a period in the module name.

### 3.8 Using extension packages

**Building and installing packages**

Gauche comes with some amount of libraries, but they aren’t enough at all to use Gauche in the production environment. There are number of additional libraries available. We call them extension packages, or simply packages. Each package usually provides one or more modules that adds extra functionality. Most of the packages provide binding to other C libraries, such as graphics libraries or database clients. If the package has some C code, it is likely that you need to compile it on your machine with the installed Gauche system.

Usually a package is in the form of compressed tarball, and the standard "ungzip + untar + configure + make + make install" sequence does the job. Read the package’s document, for you may be able to tailor the library for your own needs by giving command-line options to the `configure` script.

From Gauche 0.8, an utility script called `gauche-package` is installed for the convenience. It automates the build and install process of packages.

Suppose you have downloaded a package `Package-1.0.tar.gz`. If the package follows the convention, all you have to do is to type this:

```shell
$ gauge-package install Package-1.0.tar.gz
```

It ungzsips and untars the package, `cd` into the `Package-1.0` subdirectory, run configure, make, and make install. By default, `gauche-package` untars the tarball in the current working directory. You can change it by a customization file; see below.
If you need a special privilege to install the files, you can use \texttt{--install-as} option which runs \texttt{make install} part via the \texttt{sudo} program.

\begin{verbatim}
$ gauche-package install --install-as=root Package-1.0.tar.gz
\end{verbatim}

If it doesn't work for you, you can just build the package by \texttt{gauche-package build Package-1.0.tar.gz}, then manually \texttt{cd} to the \texttt{Package-1.0} directory and run \texttt{make install}.

You can give configuration options via \texttt{-C} or \texttt{--configure-options} command-line argument, like this:

\begin{verbatim}
$ gauche-package install -C "--prefix=/usr/local" Package-1.0.tar.gz
\end{verbatim}

If the package has adopted the new package description file, it can remember the configuration options you have specified, and it will automatically reuse them when you install the package again. (If you’re a package developer, check out \texttt{examples/spigot/README} file in the Gauche source tree to see how to cooperate with Gauche’s package management system.)

If you don’t have a tarball in your local directory, but you know the URL where you can download it, you can directly give the URL to \texttt{gauche-package}. It understands \texttt{http} and \texttt{ftp}, and uses either \texttt{wget} or \texttt{ncftpget} to download the tarball, then runs configure and make.

\begin{verbatim}
$ gauche-package install http://www.example.com/Package-1.0.tar.gz
\end{verbatim}

**Customizing \texttt{gauche-package}**

The \texttt{gauche-package} program reads \texttt{~/.gauche-package} if it exists. It must contain an associative list of parameters. It may look like this:

\begin{verbatim}
( (build-dir : "~/.home/shiro/tmp")
  (gzip : /usr/local/bin/gzip")
  (bzip2 : /usr/local/bin/bzip2")
  (tar : /usr/local/bin/gtar")
)
\end{verbatim}

The following is a list of recognized parameters. If the program isn’t given in the configuration file, \texttt{gauche-package} searches \texttt{PATH} to find one.

- \texttt{build-dir}
  
  A directory where the tarball is extracted. If URL is given, the downloaded file is also placed in this directory.

- \texttt{bzip2}
  Path to the program \texttt{bzip2}.

- \texttt{cat}
  Path to the program \texttt{cat}.

- \texttt{make}
  Path to the program \texttt{make}.

- \texttt{ncftpget}
  Path to the program \texttt{ncftpget}.

- \texttt{rm}
  Path to the program \texttt{rm}.

- \texttt{sudo}
  Path to the program \texttt{sudo}.

- \texttt{tar}
  Path to the program \texttt{tar}.

- \texttt{wget}
  Path to the program \texttt{wget}.

### 3.9 Building standalone executables

When you want to distribute your Gauche scripts or applications, the users need to install Gauche runtime on their machine. Although it is always the case for any language implementations—you need Java runtime to run Java applications, or C runtime to run C applications—it may be an extra effort to ask users to install not-so-standard language runtimes.

To ease distribution of Gauche applications, you can create a stand-alone executable. It statically links entire Gauche system so that it runs by just copying the executable file.
Quick recipe

To generate a standalone executable, just give your script file to the tools/build-standalone script, which is installed as a part of Gauche.

```
gosh tools/build-standalone yourscript.scm
```

It will create an executable file yourscript (or yourscript.exe on Windows) in the current directory.

To specify the output name different from the script name, give -o option:

```
gosh tools/build-standalone -o yourcommand yourscript.scm
```

When your script needs supporting library files, you should list those files as well:

```
gosh tools/build-standalone yourscript.scm lib/library1.scm lib/library2.scm
```

The library file paths need to be relative to the respective load path. See the explanation of -I option below.

Catches

There are a few things you should be aware of.

- The size of the binary tend to be large, since it contains the entire Gauche system regardless of whether your application use it or not. You can strip down the size if you need to, but you need to rebuild Gauche library to do so. See doc/HOWTO-standalone.txt in the source tree for the details.
- The generated binary still depends on external dynamically linked libraries, such as libpthread. The exact dependency may differ how Gauche is configured, and can be checked by running system-provided tools, such as ldd on most Unix systems and MinGW or otool -L on OSX, on the generated standalone binaries. You may want to ensure the users have required libraries.
- Currently we don’t yet have a convenient way to statically link extension libraries. We’re working on it.
- If Gauche is configured to use gdbm, it is linked to the standalone binary by default, hence the binary itself is covered by GPL. In case if you need to distribute binaries under BSD license, you need to give -D GAUCHE_STATIC_EXCLUDE_GDBM flag to tools/build-standalone. It makes tools/build-standalone not to link gdbm (and your script won’t be able to use it).
- If you build Gauche with mbedtls TLS support (if you have libmbedtls on your machine, Gauche include its support by default), the resulting standalone binary also depends on libmbedtls DSO files. If you’re not sure mbedtls TLS DSO files are available on target machines, you can exclude rfc.tls.mbed module by giving -D GAUCHE_STATIC_EXCLUDE_MBED flag to tools/build-standalone.

Using build-standalone

```
gosh tools/build-standalone [options] script-file [library-file ...]
```

Create a stand-alone binary to execute a Gauche program in script-file. It is executed as if it is run as gosh script-file, with a few differences.

The main thing is that since script-file is no longer loaded from file, code that references paths relative to script-file won’t work. One example is (add-load-path dir :relative) (see Section 6.22.1 [Loading Scheme file], page 286). Auxiliary library files required by script-file must be explicitly listed as library-file ..., so that they are bundled together into the executable.

The following command-line options are recognized.
**-o outfile**  
Specifies output executable filename. When omitted, the basename of `script-file` without extension is used. (Or, on Windows, swapping extension with `.exe`).

**-D var [=val]**  
Add C preprocessor definitions while compiling the generated C code. An important use case of this option is to exclude gdbm dependency from the generated binaries, by specifying `-D GAUCHE_STATIC_EXCLUDE_GDBM`. Note that you need a whitespace between `-D` and `var`. 
This option can be specified multiple times.

**-I load-path**  
Specifies the load path where `library-file` ... are searched. The names given to `library-file` must match how they are loaded or used. If such paths are not relative to the directory you run `tools/build-standalone`, you have to tell where to find those libraries with this option.
For example, suppose you have this structure:

```
project/src/
  +----- main.scm
  |     (use myscript.util)
  +----- myscript/util.scm
      (define-module myscript.util ...)
```

If you run `tools/build-standalone` in the directory as `src`, you can just say this:

```
gosh tools/build-standalone main.scm myscript/util.scm
```

But if you run it under `project`, you need to say this:

```
gosh tools/build-standalone -I src src/main.scm myscript/util.scm
```

Another example; you have a separate library directory:

```
project/
  +----- src/main.scm
  |     (use myscript.util)
  +----- lib/myscript/util.scm
      (define-module myscript.util ...)
```

If you run `tools/build-standalone` in `src`, you say this:

```
gosh tools/build-standalone main.scm myscript/util.scm
```

Or, if you run it in `project`, you say this:

```
gosh tools/build-standalone -I .. /lib main.scm myscript/util.scm
```

This option can be specified multiple times.

**--header-dir dir**  
**--library-dir dir**
These tells `tools/build-standalone` where to find Gauche C headers and static libraries. If you’ve installed Gauche on your system, `tools/build-standalone` automatically finds these from the installed directory and you don’t need to worry about them. Use these option only when you need to use Gauche runtime that’s not installed.
4 Core syntax

4.1 Lexical structure

Gauche extends R7RS Scheme lexical parser in some ways. Besides, because of historical reasons, a few of the default lexical syntax may conflict R7RS specification. You can set a reader mode to make it R7RS compliant.

Hash-bang directives

Tokens beginning with #! may have special meanings to the reader. R7RS defines two of such directives—#!fold-case and #!no-fold-case, which switches whether symbols are read in case-folding or non-case-folding mode, respectively. See Section 4.1.2 [Hash-bang token], page 46, below, for all the directives Gauche has.

Square brackets

Gauche adopts the R6RS syntax that regards [] the same as (). Both kind of parentheses are equivalent, but the kind of corresponding open and close parentheses must match. Some seasoned Lisper may frown on them, but it helps visually distinguish different roles of parentheses.

A general convention is to use [] for groupings other than function and macro application. If such grouping nests, however, use () for outer groupings. Examples:

```
(cond [(test1 x) (y z)]
      [(test2 x) (s t)]
      [else (u v)])
```

```
(let ([x (foo a b)]
      [y (bar c d)])
  (baz x y))
```

It is purely optional, so you don’t need to use them if you don’t like them. R7RS doesn’t adopt this syntax and leaves [] for extensions, so it is safe to stick to () in portable R7RS programs. (If the reader is in strict-r7 mode, an error is signalled when [] is used. See Section 6.21.7.2 [Reader lexical mode], page 272, for the details.)

Scheme-specific modes of some editors (e.g. Quack on Emacs) allows you to type just ) and inserts either ] or ) depending on which kind parenthesis it is closing. We recommend using such modes if you use this convention.

Symbol names

Symbol names are case sensitive by default (see Section 2.4 [Case-sensitivity], page 15). Symbol name can begin with digits, ’+’ or ’-’, as long as the entire token doesn’t consist valid number syntax. Other weird characters can be included in a symbol name by surrounding it with ’|’, e.g. ’|this is a symbol|’. See Section 6.7 [Symbols], page 155, for details.

Numeric literals

Either integral part or fraction part of inexact real numbers can be omitted if it is zero, i.e. 30., .25, -.4 are read as real numbers. The number reader recognizes ’#’ as insignificant digits. Complex numbers can be written both in the rectangular format (e.g. 1+0.3i) and in the polar format (e.g. 3.0@1.57). Inexact real numbers include the positive infinity, the negative infinity, and NaN, which are represented as +inf.0, -inf.0 and +nan.0, respectively. (-nan.0 is also read as NaN.)
Gauche supports SRFI-169 (underscores in numbers) notation; you can insert a character _ between digits in numeric literals to improve readability, e.g. \#b1100_1010_1111_1110. A valid underscore must be surrounded by digits; 1_2_3 is ok, but _123, 123_, and 12__3 are not.

Gauche also adopts Common-Lisp style radix prefixed numeric literals, e.g. \#3r120 (120 in base-3, 15 in decimal). Radix between 2 and 36 are recognized; alphabetic letters a-zA-Z are used beyond decimal.

For the polar notation of complex numbers, Gauche allows the suffix \pi to denote the phase by multiples of \pi. The Scheme syntax use radians for the phase, but you can only approximate \pi with the floating point numbers, so it can’t represent round numbers except zero angle.

\begin{verbatim}
  gosh> 2@3.141592653589793
  -2.0+2.4492935982947064e-16i
\end{verbatim}

With the \pi suffix, you can get a round numbers.

\begin{verbatim}
  gosh> 2@1\pi
  -2.0
  gosh> 2@0.5\pi
  0.0+2.0i
  gosh> 2@-0.5\pi
  0.0-2.0i
\end{verbatim}

\textit{Hex character escapes}

You can denote a character using hexadecimal notation of the character code in some literals; specifically, character literals, character set literals, string literals, symbols, regular expression literals.

R7RS adopted a hex escape notation \xNNNN; for strings and symbols surrounded by vertical bars, and \#\xNNNN for characters. The number of digits is variable, and the character code is Unicode codepoint.

Gauche had been using two types of escapes: \u and \x. In general, \u is for Unicode codepoint, while \x is for the character code in the internal encoding. Besides, except character literals, we used fixed number of digits, instead of using the terminator ; as in R7RS.

Since 0.9.4, we interpret \x-escape as R7RS whenever if it consists a valid R7RS hex-escape, and if not, try to interpret it as legacy Gauche hex-escape.

Although rarely, there are cases that can interpreted both in R7RS syntax and legacy Gauche syntax, but yielding different characters. Reading legacy files with such literals in the current Gauche may cause unexpected behavior. You can switch the reader mode so that it becomes backward-compatible. See Section 6.21.7.2 [Reader lexical mode], page 272, for the details.

\textit{Extended sharp syntax}

Many more special tokens begins with ‘#’ are defined. See the table below.

\subsection{4.1.1 Sharp syntax}

The table below lists sharp-syntaxes.

\begin{itemize}
  \item \texttt{#!} \hspace{1cm} \texttt{[R6RS][R7RS][SRFI-22]} It is either a beginning of an interpreter line (shebang) of a script, or a special token that affects the mode of the reader. See ‘hash-bang token’ section below.
  \item \texttt{#"} \hspace{1cm} Introduces an interpolated string. See Section 6.11.4 [String interpolation], page 173.
\end{itemize}
## Chapter 4: Core syntax

Unused.

[R7RS] Introduces a vector.

Unused.

Bitvector or an incomplete string. See Section 6.11 [Strings], page 170.

Unused.

[SRFI-10] Introduces reader constructor syntax.

Unused.

Introduces a literal regular expression. See Section 6.12 [Regular expressions], page 183.

[R7RS] Introduces a vector.

[Introduces a literal character set. See Section 6.10 [Character sets], page 165.

[Introduces a literal character. See Section 6.9 [Characters], page 160.

[Introduces a literal character. See Section 6.7 [Symbols], page 155.


Introduces an unreadable object.

Unused.

Introduces debug macros. See Section 3.4 [Debugging], page 33.

Unused.

[R7RS] Binary number prefix.

Unused.

[R7RS] Decimal number prefix.

[R7RS] Exact number prefix.

[R7RS] Boolean false, or introducing R7RS uniform vector. See Section 6.13.2 [Uniform vectors], page 198. R7RS defines both #f and #false as a boolean false value.

Unused.

[R7RS] Inexact number prefix.

Unused.

[R7RS] Octal number prefix.

Unused.

[R7RS vector.@] introducing R7RS uniform vector. See Section 6.13.2 [Uniform vectors], page 198.

[R7RS] Boolean true. R7RS defines #t and #true as a boolean true value.

[R7RS vector.@] introducing R7RS uniform vector. See Section 6.13.2 [Uniform vectors], page 198. R7RS uses #u8 prefix for bytevectors, which is compatible to u8 uniform vectors.

Unused.

[R7RS] Hexadecimal number prefix.

Unused.

Introduces a literal character set. See Section 6.10 [Character sets], page 165.

[R7RS] Introduces a literal character. See Section 6.9 [Characters], page 160.
#], #^, _

Unused.

#' Legacy syntax for string interpolation, superseded by "."

#{

Unused.

#] [SRFI-30] Introduces a block comment. Comment ends by matching "]#".

Unused.

### 4.1.2 Hash-bang token

A character sequence #! has two completely different semantics, depending on how and where it occurs.

If a file begins with #!/ or #! (hash, bang, and a space), then the reader assumes it is an interpreter line (shebang) of a script and ignores the rest of characters until the end of line. (Actually the source doesn’t need to be a file. The reader checks whether it is the beginning of a port.)

Other than the above case, #!identifier is read as a token with special meanings. This kind token can be a special directive for the reader, instead of read as a datum.

By default, the following tokens are recognized.

#!fold-case
#!no-fold-case

Switches the reader’s case sensitivity; #!fold-case makes the reader case insensitive, and #!no-fold-case makes it case sensitive. (Also see Section 2.4 [Case-sensitivity], page 15).

#!r6rs

This token is introduced in R6RS and used to indicate the program strictly conforms R6RS. Gauche doesn’t conform R6RS, but currently it just issues warning when it sees #!r6rs token, and it keeps reading on.

#!r7rs

Make the reader strict-r7 mode, that compiles R7RS. See Section 6.21.7.2 [Reader lexical mode], page 272, for the details.

#!gauche-legacy

Make the reader legacy mode, that is compatible to Gauche 0.9.3 and before. See Section 6.21.7.2 [Reader lexical mode], page 272, for the details.

### 4.2 Literals

quote datum

[Special Form]

[R7RS base] Evaluates to datum.

(quote x) ⇒ x
(quote (1 2 3)) ⇒ (1 2 3)

'datum

[Reader Syntax]

[R7RS] Equivalent to (quote datum).

'x ⇒ x
'(1 2 3) ⇒ (1 2 3)

Note: Literals are immutable. You’ll get an error if you try to change, for example, a quoted pair with set-car! or a literal string with string-set!. Mutability may be managed differently for each type, so some object may not raise an error even if it appears in a part of literals (it is often the case with user-defined class with read-time constructor, see Section 6.21.7.3 [Read-time constructor], page 273. However, if you ever modify a literal data, it is not guaranteed that the program runs correctly.
4.3 Making procedures

\textbf{lambda} \textit{formals body} ... \quad \text{[Special Form]}
\textbf{^} \textit{formals body} ... \quad \text{[Special Form]}

[R7RS+] Evaluates to a procedure. The environment in effect when this expression is evaluated is stored in the procedure. When the procedure is called, \textit{body} is evaluated sequentially in the stored environment extended by the bindings of the formal arguments, and returns the value(s) of the last expression in the body.

^ is a concise alias of \textbf{lambda}. It is Gauche’s extension.

\[(\textbf{lambda} (a b) (+ a b))\]
\Rightarrow \text{procedure that adds two arguments}

\[((\textbf{lambda} (a b) (+ a b)) 1 2) \Rightarrow 3\]

\[((^)(a b) (+ a b)) 1 2) \Rightarrow 3\]

Gauche also extends R7RS \textbf{lambda} to take extended syntax in \textit{formals} to specify optional and keyword arguments easily. The same functionality can be written in pure R7RS, with parsing variable-length arguments explicitly, but the code tends to be longer and verbose. It is recommended to use extended syntax unless you’re writing portable code.

\textbf{Formals} should have one of the following forms:

- (\textit{variable} ...): The procedure takes a fixed number of arguments. The actual arguments are bound to the corresponding variables.

\[((\textbf{lambda} (a) a) 1) \Rightarrow 1\]

\[((\textbf{lambda} (a) a) 1 2) \Rightarrow \text{error – wrong number of arguments}\]

- \textit{variable}: The procedure takes any number of arguments. The actual arguments are collected to form a new list and bound to the variable.

\[((\textbf{lambda} a a) 1 2 3) \Rightarrow (1 2 3)\]

- (\textit{variable}_0 ... \textit{variable}_N-1. \textit{variable}_N): The procedure takes at least \(N\) arguments. The actual arguments up to \(N\) is bound to the corresponding variables. If more than \(N\) arguments are given, the rest arguments are collected to form a new list and bound to \textit{variable}_N.

\[((\textbf{lambda} (a b . c) (\text{print} \ "a"= a " b"= b " c"= c)) 1 2 3 4 5)\]
\Rightarrow \text{prints a=1 b=2 c=(3 4 5)}

- (\textit{variable} ... \textit{extended-spec} ...): Extended argument specification. Zero or more variables that specifies required formal arguments, followed by an \textit{extended spec}, a list beginning with a keyword :\textit{optional}, :\textit{key} or :\textit{rest}.

The \textit{extended-spec} part consists of the optional argument spec, the keyword argument spec and the rest argument spec. They can appear in any combinations.

:optional \textit{optspec} ...

Specifies optional arguments. Each \textit{optspec} can be either one of the following forms:

\textit{variable}
\textit{(variable init-expr)}

The \textit{variable} names the formal argument, which is bound to the value of the actual argument if given, or the value of the expression \textit{init-expr} otherwise. If \textit{optspec} is just a variable, and the actual argument is not given to it, then \textit{variable} will be bound to \#<\texttt{undef}> (see Section 6.5 [Undefined values], page 140).
The expression `init-expr` is only evaluated if the actual argument for `variable` is not given. The scope in which `init-expr` is evaluated includes the preceding formal arguments.

(((lambda (a b :optional (c (+ a b))) (list a b c))
  1 2)  ⇒  (1 2 3)

(((lambda (a b :optional (c (+ a b))) (list a b c))
  1 2  -1)  ⇒  (1 2 -1)

(((lambda (a b :optional c) (list a b c))
  1 2)  ⇒  (1 2 #<undef>)

(((lambda (:optional (a 0) (b (+ a 1))) (list a b))
  )  ⇒  (0 1)

The procedure signals an error if more actual arguments than the number of required and optional arguments are given, unless it also has `:key` or `:rest` argument spec.

(((lambda (:optional a b) (list a b)) 1 2 3)
  ⇒  error - too many arguments

(((lambda (:optional a b :rest r) (list a b r)) 1 2 3)
  ⇒  (1 2 (3))

`:key keyspec ... [:allow-other-keys [variable]]`

Specifies keyword arguments. Each `keyspec` can be either one of the following forms.

  variable
  (variable init-expr)
  ((keyword variable) init-expr)

The `variable` names the formal argument, which is bound to the actual argument given with the keyword of the same name as `variable`. When the actual argument is not given, `init-expr` is evaluated and the result is bound to `variable` in the second and third form, or `#<undef>` is bound in the first form.

(define f (lambda (a :key (b (+ a 1))) (c (+ b 1)))
  (list a b c)))

  (f 10)  ⇒  (10 11 12)
  (f 10 :b 4)  ⇒  (10 4 5)
  (f 10 :c 8)  ⇒  (10 11 8)
  (f 10 :c 1 :b 3)  ⇒  (10 3 1)

With the third form you can name the formal argument differently from the keyword to specify the argument.

(((lambda (:key (((aa a) -1)) :aa) :aa 2)
  ⇒  2

By default, the procedure with keyword argument spec raises an error if a keyword argument with an unrecognized keyword is given. Giving `:allow-other-keys` in the formals suppresses this behavior. If you give `variable` after `:allow-other-keys`, the list of unrecognized keywords and their arguments are bound to it. Again, see the example below will help to understand the behavior.
((lambda (:key a) a)
 :a 1 :b 2) ⇒ error - unknown keyword :b

((lambda (:key a :allow-other-keys) a)
 :a 1 :b 2) ⇒ 1

((lambda (:key a :allow-other-keys z) (list a z))
 :a 1 :b 2) ⇒ (1 ( :b 2))

When used with :optional argument spec, the keyword arguments are searched after all the optional arguments are bound.

((lambda (:optional a b :key c) (list a b c))
 1 2 :c 3) ⇒ (1 2 3)

((lambda (:optional a b :key c) (list a b c))
 :c 3) ⇒ (:c 3 #<undef>)

((lambda (:optional a b :key c) (list a b c))
 1 :c 3) ⇒ error - keyword list not even

:rest variable

Specifies the rest argument. If specified without :optional argument spec, a list of remaining arguments after required arguments are taken is bound to variable. If specified with :optional argument spec, the actual arguments are first bound to required and all optional arguments, and the remaining arguments are bound to variable.

((lambda (a b :rest z) (list a b z))
 1 2 3 4 5) ⇒ (1 2 (3 4 5))

((lambda (a b :optional c d :rest z) (list a b c d z))
 1 2 3 4 5) ⇒ (1 2 3 4 (5))

((lambda (a b :optional c d :rest z) (list a b c d z))
 1 2 3) ⇒ (1 2 3 #<undef> ()

When the rest argument spec is used with the keyword argument spec, both accesses the same list of actual argument—the remaining arguments after required and optional arguments are taken.

((lambda (:optional a :rest r :key k) (list a r k))
 1 :k 3) ⇒ (1 ( :k 3) 3)

See also let-optionals*, let-keywords and let-keywords* macros in Section 6.15.4 [Optional argument parsing], page 222, for an alternative way to receive optional/keyword arguments within the spec of R7RS.

`c body` . . .

A shorthand notation of (lambda (c) body ...). where c can be any character in #\[a-z\].

(map (`x (* x x)) '(1 2 3 4 5)) ⇒ (1 4 9 16 25)

cut expr-or-slot expr-or-slot2 . . .

cute expr-or-slot expr-or-slot2 . . .

[SRFI-26] Convenience macros to notate a procedure compactly. This form can be used to realize partial application, a.k.a sectioning or projection.

Each expr-or-slot must be either an expression or a symbol <>, indicating a 'slot'. The last expr-or-slot can be a symbol <...>, indicating a 'rest-slot'. Cut expands into a lambda form
that takes as many arguments as the number of slots in the given form, and whose body is an expression

\[(\text{expr-or-slot} \ \text{expr-or-slot2} \ ...)\]

where each occurrence of \(<\to\) is replaced to the corresponding argument. In case there is a rest-slot symbol, the resulting procedure is also of variable arity, and all the extra arguments are passed to the call of \text{expr-or-slot}. See the fourth example below.

\[
\begin{align*}
\text{(cut cons (+ a 1) <>)} & \equiv (\lambda(x2) (\text{cons} (+ a 1) x2)) \\
\text{(cut list 1 <> 3 <> 5)} & \equiv (\lambda(x2 \ x4) (\text{list} 1 x2 3 x4 5)) \\
\text{(cut list)} & \equiv (\lambda() (\text{list})) \\
\text{(cut list 1 <> 3 <...>)} & \equiv (\lambda(x2 . xs) (\text{apply} \ 1 x2 3 xs)) \\
\text{(cut <> a b)} & \equiv (\lambda(f) (f \ a \ b))
\end{align*}
\]

`; Usage

\[
\begin{align*}
\text{(map (cut * 2 <>) '(1 2 3 4))} \\
\text{(for-each (cut write <> port) exprs)}
\end{align*}
\]

\text{Cute} is a variation of \text{cut} that evaluates \text{expr-or-slots} before creating the procedure.

\[
\begin{align*}
\text{(cute cons (+ a 1) <>)} & \equiv (\text{let} ((xa (+ a 1))) (\lambda(x2) (\text{cons} xa x2)))
\end{align*}
\]

Gauche provides a couple of different ways to write partial applications concisely; see the \$ macro below, and also the \text{pa$} procedure (see Section 6.15.3 [Combinators], page 220).

\[
\text{\$ arg ...}
\]

A macro to chain applications, hinted from Haskell’s \$ operator (although the meaning is different). Within the macro arguments \$ arg ..., \$ delimits the last argument. For example, the following code makes the last argument for the procedure \(f\) to be \((g \ c \ d ...)\)

\[
\begin{align*}
\text{(\$ f a b \$ g c d ...)} & \equiv (f \ a \ b \ (g \ c \ d ...))
\end{align*}
\]

The \$ notation can be chained.

\[
\begin{align*}
\text{(\$ f a b \$ g c d \$ h e f ...)} & \equiv (f \ a \ b \ (g \ c \ d \ (h \ e \ f ...)))
\end{align*}
\]

If \$* appears in the argument list instead of \$, it fills the rest of the arguments, instead of just the last argument.

\[
\begin{align*}
\text{(\$ f a b \$* g c d ...)} & \equiv (\text{apply} \ f \ a \ b \ (g \ c \ d ...))
\end{align*}
\]

\[
\begin{align*}
\text{(\$ f a b \$* g h \$* hh ...)} & \equiv (\text{apply} \ f \ a \ b \ (g \ (\text{apply} \ h \ (hh ...))))
\end{align*}
\]

Furthermore, if the argument list ends with \$ or \$*, the whole expression becomes a procedure expecting the last argument(s).

\[
\begin{align*}
\text{(\$ f a b \$ g c d \$ h e f \$*)} & \equiv (\lambda(\text{arg}) (f \ a \ b \ (g \ c \ d \ (h \ e \ f \ \text{arg})))
\end{align*}
\]

The more functional the code becomes, the more you are tempted to write it as a chain of nested function calls. Scheme’s syntax can get awkward in such code. Close parentheses tend
to clutter at the end of expressions. Inner applications tends to pushed toward right columns with the standard indentation rules. Compare the following two code functionally equivalent to each other:

\[
\text{(intersperse ":"} \\
\text{(map transform-it} \\
\text{(delete-duplicates (map cdr} \\
\text{(group-sequence input))))})
\]

\[
($\text{intersperse ":"} \\
$\text{map transform-it} \\
$\text{delete-duplicates} \\
$\text{map cdr} $\text{group-sequence input})
\]

It is purely a matter of taste, and also this kind of syntax sugars can be easily abused. Use with care, but it may work well if used sparingly, like spices.

As a corner case, if neither $ nor $* appear in the argument list, it just calls the function. It is useful when the function has long name and you don’t want to indent arguments too further right.

\[
($ f$ a b c) \equiv (f a b c)
\]

case-lambda clause ...  
\[\text{Macro}\]
[R7RS case-lambda] Each clause should have the form (formals expr ...), where formals is a formal arguments list as for lambda.

This expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as procedures resulting from lambda expressions. When the procedure is called with some arguments, then the first clause for which the arguments agree with formals is selected, where agreement is specified as for the formals of a lambda expression. The variables of formals are bound to the given arguments, and the expr ... are evaluated within the environment.

It is an error for the arguments not to agree with the formals of any clause.

\[
\text{(define f} \\
\text{(case-lambda} \\
\text{[()] 'zero] \\
\text{[(a) '(one ,a)] \\
\text{[(a b) ' (two ,a ,b)]})}
\]

\[
(f) \Rightarrow \text{zero} \\
(f 1) \Rightarrow \text{(one 1)} \\
(f 1 2) \Rightarrow \text{(two 1 2)} \\
(f 1 2 3) \Rightarrow \text{Error: wrong number of arguments to case lambda}
\]

\[
\text{(define g} \\
\text{(case-lambda} \\
\text{[()] 'zero] \\
\text{[(a) '(one ,a)] \\
\text{[(a . b) ' (more ,a ,@b)]})}
\]

\[
(g) \Rightarrow \text{zero} \\
(g 1) \Rightarrow \text{(one 1)} \\
(g 1 2 3) \Rightarrow \text{(more 1 2 3)}
\]

Note that the clauses are examined sequentially to match the number of arguments, so in the following example g2 never returns (one ...).
(define g2
  (case-lambda
    [() 'zero]
    [(a . b) '(more ,a ,@b)]
    [(a) '(one ,a)])

(g2 1) ⇒ (more 1)

4.4 Assignments

(set! symbol expression) [Special Form]
(set! (proc arg . . .) expression) [Special Form]

[R7RS+ base][SRFI-17] First, expression is evaluated. In the first form, the binding of symbol is modified so that next reference of symbol will return the result of expression. If symbol is not locally bound, the global variable named symbol must already exist, or an error is signaled.

The second form is a “generalized set!” specified in SRFI-17. It is a syntactic sugar of the following form.

  ((setter proc) arg ... expression)

Note the order of the arguments of the setter method differs from CommonLisp’s setf.

Some examples:

  (define x 3)
  (set! x (list 1 2))
  x ⇒ (1 2)

  (set! (car x) 5)
  x ⇒ (5 2)

(set!-values (var . . .) expr) [Macro]
(set!-values (var ... varn) expr) [Macro]
(set!-values var expr) [Macro]

[SRFI-210] Sets values of multiple variables at once. Expr must yield as many values as var .... Each value is set to the corresponding var.

  (define a 0)
  (define b 1)
  (define c 2)
  (set!-values (a b) (values 3 4))
  a ⇒ 3
  b ⇒ 4
  (set!-values (a b) (values b a))
  a ⇒ 4
  b ⇒ 3
  (set!-values (a b . c) (values 1 2 3 4))
  a ⇒ 1
  b ⇒ 2
  c ⇒ (3 4)

See also define-values (Section 4.10 [Definitions], page 68).

(setter proc) [Function]
[SRFI-17] Returns a setter procedure associated to the procedure proc. If no setter is associated to proc, its behavior is undefined.
A setter procedure $g$ of a procedure $f$ is such that when used as $(g\ a\ b\ \ldots\ v)$, the next evaluation of $(f\ a\ b\ \ldots)$ returns $v$.

To associate a setter procedure to another procedure, you can use the setter of setter, like this:

$$\text{(set! (setter } f \text{) } g)$$

A procedure’s setter can be “locked” to it. System default setters, like set-car! for car, is locked and can’t be set by the above way. In order to lock a setter to a user defined procedure, use getter-with-setter below.

If proc is not a procedure, a setter generic function of object-apply is returned; it allows the applicable object extension to work seamlessly with the generalized set!. See Section 6.15.6 [Applicable objects], page 225, for the details.

has-setter? proc

Returns #t if a setter is associated to proc.

getter-with-setter get set

[SRFI-17] Takes two procedure get and set. Returns a new procedure which does the same thing as get, and its setter is locked to set.

The intention of this procedure is, according to the SRFI-17 document, to allow implementations to inline setters efficiently. Gauche hasn’t implement such optimization yet.

A few macros that adopts the same semantics of generalized set! are also provided. They are built on top of set!.

push! place item

Conses item and the value of place, then sets the result to place. place is either a variable or a form (proc arg . . . ), as the second argument of set!. The result of this form is undefined.

(define x (list 2))
(push! x 3)
x ⇒ (3 2)

(push! (cdr x) 4)
x ⇒ (3 4 2)

When place is a list, it roughly expands like the following.

(push! (foo x y) item)
≡
(let ((tfoo foo)
       (tx x)
       (ty y))
   ((setter tfoo) tx ty (cons item (tfoo tx ty))))

Note: Common Lisp’s push macro takes its argument reverse order. I adopted this order since it is consistent with other destructive operations. Perl’s push function takes the same argument order, but it appends item at the end of the array (Perl’s unshift is closer to push!). You can use a queue (see Section 12.19 [Queue], page 837) if you need a behavior of Perl’s push.

push-unique! place item [equal]

The value in place must be a list. If item isn’t already in the value in place, place is updated with the cons of item and its previous value. If item is already in the value, nothing is done. The equivalence is checked with equal procedure. If omitted, eqv? is assumed.

This is similar to Common Lisp’s pushnew, but the argument order is flipped.
See also `enqueue-unique!` and `queue-push-unique!` in `data.queue` (Section 12.19 [Queue], page 837).

```lisp
(define v (vector '"a"))

(push-unique! (vector-ref v 0) "b")
⇒ #(("b" "a"))

(push-unique! (vector-ref v 0) "A" string-ci=?)
⇒ #(("b" "a"))

(pop! place)  [Macro]
Retrieves the value of `place`, sets its `cdr` back to `place` and returns its `car`.

```lisp
(define x (list 1 2 3))
(pop! x) ⇒ 1
x ⇒ (2 3)

(define x (vector (list 1 2 3)))
(x ⇒ #((1 2 3))
(pop! (vector-ref x 0)) ⇒ 1
x ⇒ #(2 3)
```

Note: This works the same as Common Lisp's `pop`. Perl's `pop` pops value from the end of the sequence; its `shift` does the same thing as `pop!`.

```lisp
inc! place :optional delta  [Macro]
dec! place :optional delta  [Macro]
Evaluates the value of `place`. It should be a number. Adds (`inc!`) or subtracts (`dec!`) `delta` to/from it, and then stores the result to `place`. The default value of `delta` is 1.
This is like Common Lisp's `incf` and `decf`, except that you can't use the result of `inc!` and `dec!`.

update! place proc  [Macro]
Generalized form of `push!` etc. `Proc` must be a procedure which takes one argument and returns one value. The original value of `place` is passed to the `proc`, then its result is set to `place`.

```lisp
(define a (cons 2 3))
(update! (car a) (lambda (v) (* v 3)))
a ⇒ (6 . 3)

(update! (cdr a) (cut - <> 3))
a ⇒ (6 . 0)
```

rotate! place0 place1 ... placeN  [Macro]
It is roughly equivalent to:

```lisp
(let ((tmp0 place0) (tmp1 place1) ... (tmpN placeN))
  (set! place0 tmp1)
  (set! place1 tmp2)
  ...
  (set! placeN tmp0))
```
However, if `place` is the form `(proc arg ...)`, then each `arg` is evaluated only once.
It can be used to swap values of two locations:

\[
\begin{align*}
&\text{(let \((x \text{ (vector 1 2)})\))} \\
&\text{(rotate! \((- x 0) \ (- x 1))} \\
&x) \Rightarrow \text{#(2 1)}
\end{align*}
\]

This is similar to Common Lisp's rotatef.

### 4.5 Conditionals

#### if

```lisp
if \text{test } \text{consequent } \text{alternative} \qquad \text{[Special Form]}$

\text{if \text{test } \text{consequent}}$

[R7RS base] Test is evaluated. If it yields a true value, consequent is evaluated. Otherwise, alternative is evaluated. If alternative is not provided, it results undefined value.

```
\[
\begin{align*}
&(\text{if (number? 3) 'yes 'no}) \Rightarrow \text{yes} \\
&(\text{if (number? #f) 'yes 'no}) \Rightarrow \text{no}
\end{align*}
\]

```

```
\begin{align*}
&(\text{let ((x '(1 . 2)))} \\
&(\text{if (pair? x)} \\
&(\text{(values (car x) (cdr x))} \\
&(\text{(values #f #f))}) \\
\Rightarrow 1 \text{ and 2})
\end{align*}
```

#### cond

```lisp
cond \ text{clause1 clause2 . . .} \qquad \text{[Special Form]}
```

[R7RS+ base][SRFI-61] Each clause must be the form

```
\[
\begin{align*}
&(\text{test expr ...}) \\
&(\text{test => expr}) \\
&(\text{test guard => expr}) \\
&(\text{else expr expr2 ...})
\end{align*}
```

The last form can appear only as the last clause.

cond evaluates test of each clauses in order, until it yields a true value. Once it yields true, if the clause is the first form, the corresponding exprs are evaluated and the result(s) of last expr is(are) returned; if the clause is the second form, the expr is evaluated and it must yield a procedure that takes one argument. Then the result of test is passed to it, and the result(s) it returns will be returned.

The third form is specified in SRFI-61. In this form, test can yield arbitrary number of values. The result(s) of test is(are) passed to guard; if it returns a true value, expr is applied with an equivalent argument list, and its result(s) is(are) returned. If guard returns #f, the evaluation proceeds to the next clause.

If no test yields true, and the last clause is not the fourth form (else clause), an undefined value is returned.

If the last clause is else clause and all tests are failed, exprs in the else clause are evaluated, and its last expr's result(s) is(are) returned.

```
\[
\begin{align*}
&(\text{cond ((> 3 2) 'greater) } \\
&((< 3 2) 'less)) \Rightarrow \text{greater} \\
&(\text{cond ((> 3 3) 'greater) } \\
&((< 3 3) 'less) \\
&(\text{else 'equal))) \Rightarrow \text{equal} \\
&(\text{cond ((assv 'b '((a 1) (b 2))) => cadr) } \\
&(\text{else #f})) \Rightarrow 2
\end{align*}
```

**Case** key-expr clause1 clause2 . . .  

[Special Form]

[R7RS+ base][SRFI-87] Key-expr may be any expression that yields one value. Each clause should have the form

```
((datum ...) expr expr2 ...)
((datum ...) => proc)
```

where each datum is an external representation of some object. All the datums must be distinct. The last clause may be an “else clause,” which has the form

```
(else expr expr2 ...)
(else => proc)
```

First, key-expr is evaluated and its result is compared against each datum. If the result of evaluating key-expr is equivalent (using `eqv?`, see Section 6.2.1 [Equality], page 110) to a datum, then the expressions in the corresponding clause are evaluated sequentially, and the result(s) of the last expression in the clause is(are) returned from the case expression. The forms containing `=>` are specified in SRFI-87. In these forms, the result of key-expr is passed to proc, and its result(s) is(are) returned from the case expression.

If the result of evaluating key-expr is different from every datum, then if there is an else clause its expressions are evaluated and the result(s) of the last is(are) the result(s) of the case expression; otherwise the result of the case expression is undefined.

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⇒ composite

(case (car '(c d))
  ((a) 'a)
  ((b) 'b)) ⇒ undefined

(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant)) ⇒ consonant

(case 6
  ((2 4 6 8) => (cut + <> 1))
  (else => (cut - <> 1))) ⇒ 7

(case 5
  ((2 4 6 8) => (cut + <> 1))
  (else => (cut - <> 1))) ⇒ 4
```

**Ecase** key-expr clause1 clause2 . . .  

[Macro]

This works exactly like case, except when there’s no else clause and the value of key-expr expression doesn’t match any of datums provided in clauses. While case form returns undefined value for such case, ecase raises an error.

```
(ecase 5 ((1) 'a) ((2 3) 'b) ((4) 'c))
⇒ ERROR: ecase test fell through: got 5, expecting one of (1 2 3 4)
```

**Typcase** key-expr clause1 clause2 . . .  

[Macro]

This is taken from Common Lisp. Key-expr is a Scheme expression yielding one value. Each clause must have either one of the following forms. The else clause can only appear as the last clause, if any.
(type-expr expr ...)
(else expr1 expr2 ...)

Each type-expr must be a type expression (see Section 6.1 [Types and classes], page 105).

This macro first evaluates key-expr, then tests if its value is of the type type-expr, in turn. If
the type constraint is satisfied, the expr ... in the clause is evaluated and the last result(s)
becomes the value of typecase. Type constraint is checked with of-type? (see Section 6.1.2
[Generic type predicates], page 105). If none of the type constraint is satisfied, exprs in the
else clause is evaluated if any, or an undefined value results.

(let ((tmp key-expr))
  (cond
    ((of-type? tmp <integer>) 'a)
    ((of-type? tmp (</> <string> <symbol>)) 'b)))

etypecase key-expr clause1 clause2 ...

Like typecase, but if the value of key-expr satisfies no type-expr and there’s no else clause,
an error is signaled. If there’s an else clause, it is the same as typecase.

(and (= 2 2) (> 2 1)) ⇒ #t
(and (= 2 2) (< 2 1)) ⇒ #f
(and 1 2 'c '(f g)) ⇒ (f g)
(and) ⇒ #t

or test ...

(1 3 0)) ⇒ (b c)

when test expr1 expr2 ...

unless test expr1 expr2 ...

[R7RS base] Evaluates test. If it yields true value (or false in case of unless), expr1 and
expr2 ... are evaluated sequentially, and the result(s) of the last evaluation is(are) returned.
Otherwise, undefined value is returned.
**assume** test-expr [message obj ...]  
[Macro]  
[SRFI-145] Evaluates test-expr and returns its value.

Also, this form declares the programmer’s intent that the code following this path always satisfy test-expr.

Currently, Gauche always signals an error if test-expr evaluates to #f.

```lisp
(define (rsqrt x)
  (assume (and (real? x) (>= x 0)))
  (sqrt x))
```

gosh> (rsqrt -1)
*** ERROR: Invalid assumption: (and (real? x) (>= x 0))

If optional arguments message obj ... are passed, they are given to the error procedure when test-expr yields false.

```lisp
(define (rsqrt x)
  (assume (and (real? x) (>= x 0))
    "Argument must be nonnegative real number, but got:" x)
  (sqrt x))
```

gosh> (rsqrt -1)
*** ERROR: Argument must be nonnegative real number, but got: -1

Note: This form is advisory—it isn’t guaranteed for an error to be signaled when test-expr fails. For example, we may add an optimization option that omits testing in speed-optimized code in future. We may also enhance the compiler to generate better code using the given information—for example, in the above real-sqrt code, the compiler could theoretically deduce that (sqrt x) only needs to work as real functions, so it would be able to generate specialized code. Use this form to inform the compiler and the reader your intention.

**assume-type** expr type [message obj ...]  
[Macro]  
Evaluates expr, and Checks if the value has type type. If not, raises an error. The result of expr is returned.

As type, you can specify a Gauche class or a descriptive type. The value is of type if it satisfies (of-type? value type) (see Section 6.1 [Types and classes], page 105).

If optional arguments message obj ... are passed, they are given to the error procedure when test-expr yields false.

```lisp
(define (strlen s)
  (assume-type s <string>)
  (string-length s))
```

gosh> (strlen 4)
*** ERROR: s is supposed to be of type #<class <string>>, but got 4

```lisp
(define (strlen s)
  (assume-type s <string> "Argument s must be a string, but got:" s)
  (string-length s))
```

gosh> (strlen 4)
*** ERROR: Argument s must be a string, but got: 4

The type assumption may be used by the compiler future compilers for optimizations. In order for the compiler to use the type constraint information, type must be an expression
statically computable at compile-time. That is, it must be either a class or a type constructor expression, or a constant binding to them.

Note: Like assume, this form is advisory; it is not guaranteed that the check is performed, nor expr is evaluated.

4.6 Binding constructs

\[
\begin{align*}
\text{let} &\ ((\text{var} \ expr) \ldots) \ body \ldots & \text{[Special Form]} \\
\text{let*} &\ ((\text{var} \ expr) \ldots) \ body \ldots & \text{[Special Form]} \\
\text{letrec} &\ ((\text{var} \ expr) \ldots) \ body \ldots & \text{[Special Form]} \\
\text{letrec*} &\ ((\text{var} \ expr) \ldots) \ body \ldots & \text{[Special Form]}
\end{align*}
\]

[R7RS base] Creates a local scope where var \ldots are bound to the value of expr \ldots, then evaluates body \ldots. Vars must be symbols, and there shouldn’t be duplicates. The value(s) of the last expression of body \ldots becomes the value(s) of this form.

The four forms differ in terms of the scope and the order exprs are evaluated. Let evaluates exprs before (outside of) let form. The order of evaluation of exprs is undefined, and the compiler may reorder those exprs freely for optimization. Let* evaluates exprs, in the order they appears, and each expr is evaluated in the scope where vars before it are bound.

Letrec evaluates exprs, in an undefined order, in the environment where vars are already bound (to an undefined value, initially). letrec is necessary to define mutually recursive local procedures. Finally, letrec* uses the same scope rule as letrec, and it evaluates expr in the order of appearance.

\[
(\text{define} \ x \ '\text{top-x})
\]

\[
\begin{align*}
(\text{let} &\ ((x 3) (y x)) \ (\text{cons} \ x \ y)) \Rightarrow (3 . \text{top-x}) \\
(\text{let*} &\ ((x 3) (y x)) \ (\text{cons} \ x \ y)) \Rightarrow (3 . 3)
\end{align*}
\]

\[
\begin{align*}
(\text{let} &\ ((\text{cons} \ (\text{lambda} \ (a b) (+ a b))) \\
&\ (\text{list} \ (\text{lambda} \ (a b) \ (\text{cons} \ a \ (\text{cons} \ b \ 0)))))) \\
(\text{list} 1 2)) \Rightarrow (1 \text{ 2 . 0})
\end{align*}
\]

\[
\begin{align*}
(\text{letrec} &\ ((\text{cons} \ (\text{lambda} \ (a b) (+ a b))) \\
&\ (\text{list} \ (\text{lambda} \ (a b) \ (\text{cons} \ a \ (\text{cons} \ b \ 0)))))) \\
(\text{list} 1 2)) \Rightarrow 3
\end{align*}
\]

You need to use letrec* if evaluation of one expr requires the value of var that appears before the expr. In the following example, calculating the value of a and b requires the value of cube, so you need letrec*. (Note the difference from the above example, where calculating the value of list doesn’t need to take the value of cons bound in the same letrec. The value of cons isn’t required until list is actually applied.)

\[
(\text{letrec*} ((\text{cube} \ (\text{lambda} \ (x) (* x x x))) \\
(a (+ (\text{cube} \ 1) (\text{cube} \ 12))) \\
(b (+ (\text{cube} \ 9) (\text{cube} \ 10))))
\]

\[
(= a b) \Rightarrow #t
\]

This example happens to work with letrec in the current Gauche, but it is not guaranteed to keep working in future. You just should not rely on evaluation order when you use letrec. In retrospect, it would be a lot simpler if we only had letrec*. Unfortunately letrec preceded for long time in Scheme history and it’s hard to remove that. Besides, letrec does have more opportunities to optimize than letrec*.

\[
\text{let1} \ \text{var} \ expr \ body \ldots
\]

[Macro]

A convenient macro when you have only one variable. Expanded as follows.
Chapter 4: Core syntax

(let ((var expr)) body ...)

if-let1 var expr then [Macro]
if-let1 var expr then else [Macro]

This macro simplifies the following idiom:

(let1 var expr
  (if var then else))

rlet1 var expr body ... [Macro]

This macro simplifies the following idiom:

(let1 var expr
  body ...
  var)

and-let* (binding ...) body ... [Macro]

[SRFI-2] In short, it works like let*, but returns #f immediately whenever the expression in bindings evaluates to #f.

Each binding should be one of the following form:

(variable expression)
The expression is evaluated; if it yields true value, the value is bound to variable, then proceed to the next binding. If no more bindings, evaluates body .... If expression yields #f, stops evaluation and returns #f from and-let*.

(expressionx)
In this form, variable is omitted. Expression is evaluated and the result is used just to determine whether we continue or stop further evaluation.

bound-variable
In this form, bound-variable should be an identifier denoting a bound variable. If its value is not #f, we continue the evaluation of the clauses.

Let’s see some examples. The following code searches key from an assoc-list alist and returns its value if found.

(and-let* ((entry (assoc key alist))) (cdr entry))

If arg is a string representation of an exact integer, returns its value; otherwise, returns 0:

(or (and-let* ((num (string->number arg))
  ( (exact? num) )
  ( (integer? num) ))
  num)
  0)

The following is a hypothetical code that searches a certain server port number from a few possibilities (environment variable, configuration file, ...)

(or (and-let* ((val (sys-getenv "SERVER_PORT")))
  (string->number val))
  (and-let* ((portfile (expand-path "~/server_port"))
    ( (file-exists? portfile) )
    (val (call-with-input-string portfile port->string)))
  (string->number val))
  8080) ; default

and-let1 var test exp1 exp2 ...

[Macro]

Evaluates test, and if it isn’t #f, binds var to it and evaluates exp1 exp2 .... Returns the result(s) of the last expression. If test evaluates to #f, returns #f.
This can be easily written by `and-let*` or `if-let1` as follows. However, we’ve written this idiom so many times that it deserves another macro.

```
(and-let1 var test
  exp1
  exp2 ...)
≡
(and-let* ([var test])
  exp1
  exp2 ...)
≡
(if-let1 var test
  (begin exp1 exp2 ...)
  #f)
```

**fluid-let** `((var val) ...) body ...`  
[Macro]  
A macro that emulates dynamic scoped variables. Vars must be variables bound in the scope including `fluid-let` form. Vals are expressions. Fluid-let first evaluates vals, then evaluates body ..., with binding vars to the corresponding values during the dynamic scope of body ....  
Note that, in multithreaded environment, the change of the value of vars are visible from all the threads. This form is provided mainly for the porting convenience. Use parameter objects instead (see Section 6.16.1 [Parameters], page 233) for thread-local dynamic state.

```
(define x 0)
(define (print-x) (print x))
(fluid-let ((x 1))
  (print-x)) ⇒ ;; prints 1
```

**receive** `formals expression body ...`  
[Special Form]  
[SRFI-8] This is the way to receive multiple values. Formals can be a (maybe-improper) list of symbols. Expression is evaluated, and the returned value(s) are bound to formals like the binding of lambda formals, then body ... are evaluated.

```
(define (divrem n m)
  (values (quotient n m) (remainder n m)))

(receive (q r) (divrem 13 4) (list q r))
⇒ (3 1)

(receive all (divrem 13 4) all)
⇒ (3 1)

(receive (q . rest) (divrem 13 4) (list q rest))
⇒ (3 (1))
```

See also `call-with-values` in Section 6.15.10 [Multiple values], page 230, which is the procedural equivalent of `receive`. You can use `define-values` (see Section 4.10 [Definitions], page 68) to bind multiple values to variables simultaneously. Also `let-values` and `let* values` below provides `let`-like syntax with multiple values.
let-values ((vars expr) ...) body ...

[Macro]

[R7RS base] vars are a list of variables. expr is evaluated, and its first return value is bound to the first variable in vars, its second return value to the second variable, and so on, then body is evaluated. The scope of exprs are the outside of let-values form, like let.

(let-values (((a b) (values 1 2))
  ((c d) (values 3 4)))
  (list a b c d)) ⇒ (1 2 3 4)

(let ((a 1) (b 2) (c 3) (d 4))
  (let-values (((a b) (values c d))
    ((c d) (values a b)))
    (list a b c d))) ⇒ (3 4 1 2)

vars can be a dotted list or a single symbol, like the lambda parameters.

(let-values (((x . y) (values 1 2 3 4)))
  y) ⇒ (2 3 4)

(let-values (((x (values 1 2 3 4)))
  x) ⇒ (1 2 3 4)

If the number of values returned by expr doesn’t match what vars expects, an error is signaled.

let*-values ((vars expr) ...) body ...

[Macro]

[R7RS base] Same as let-values, but each expr’s scope includes the preceding vars.

(let ((a 1) (b 2) (c 3) (d 4))
  (let*-values (((a b) (values c d))
    ((c d) (values a b)))
    (list a b c d))) ⇒ (3 4 3 4)

rec var expr

rec (name . vars) expr ...


In the first form, evaluates expr while var in expr is bound to the result of expr. The second form is equivalent to the followings.

(rec name (lambda vars expr ...))

Some examples:

;; constant infinite stream
(rec s (cons 1 (delay s)))

;; factorial function
(rec (f n)
  (if (zero? n)
    1
    (* n (f (- n 1)))))

4.7 Grouping

begin form ...

[Special Form]

[R7RS base] Evaluates forms sequentially, and returns the last result(s).

Begin doesn’t introduce new scope like let, that is, you can’t place "internal define" at the beginning of forms generally. Semantically begin behaves as if forms are spliced into the
surrounding context. For example, toplevel expression like the following is the same as two
toplevel definitions:

\[
\text{(begin (define x 1) (define y 2))}
\]

Here’s a trickier example:

\[
\text{(let ()}
\text{ (begin}
\text{ (define x 2)
\text{ (begin
\text{ (define y 3)
\text{ )})
\text{ (+ x y))
\text{ =}
\text{ (let ()
\text{ (define x 2)
\text{ (define y 3)
\text{ (+ x y)})
\text{ begin0 } exp0 exp1 \ldots \quad \text{[Macro]}
\text{ Evaluates } exp0, \text{ exp1}, \ldots, \text{ then returns the result(s) of } exp0. \text{ The name is taken from MzScheme. This is called } \text{prog1} \text{ in CommonLisp.}
\text{ Unlike } \text{begin}, \text{ this } \text{does} \text{ creates a new scope, for the } \text{begin0} \text{ form is expanded as follows.}
\text{ (receive tmp exp0 exp1 \ldots (apply values tmp))}
\text{ independently } exp \ldots \quad \text{[Macro]}
\text{ [SRFI-236] Evaluates } exp \ldots \text{ in an unspecified order, and returns an undefined result. This form is defined in SRFI-236.}
\text{ You can use } \text{begin} \text{ to group multiple expressions, but } \text{begin} \text{ also specifies the order of evaluation even when it’s unnecessary. Using } \text{independently}, \text{ you can express the intention to the readers that the order does not matter, and also gives the compiler more opportunities for optimization.}
\text{ As the order doesn’t matter, SRFI-236 does not specify the return value of this form. Gauche returns } \#<\text{undef}>, \text{ but the caller shouldn’t rely on it.}

### 4.8 Iteration

\[
\text{do ((variable init [step]) \ldots (test expr \ldots) body \ldots \quad \text{[Special Form]}} \quad \text{[R7RS base]}
\]

1. Evaluates \text{init} \ldots and binds \text{variable} \ldots to each result. The following steps are evaluated under the environment where \text{variables} are bound.
2. Evaluate test. If it yields true, evaluates expr \ldots and returns the result(s) of last expr.
3. Otherwise, evaluates body \ldots for side effects.
4. Then evaluates step \ldots and binds each result to a fresh variable \ldots, and repeat from the step 2.

The following example loops 10 times while accumulating each value of i to j and returns it.

\[
\text{(do ((i 0 (+ i 1)))}
\]
(j 0 (+ i j)))
((= i 10) j)
(print j))
⇒ 45 ; also prints intermediate values of j

If step is omitted, the previous value of variable is carried over. When there’s no expr, the
non-false value returned by test becomes the value of the do expression.

Since do syntax uses many parentheses, some prefer using square brackets as well as paren-
theses to visually distinguish the groupings. A common way is to group each variable binding,
and the test clause, by square brackets.

\[
(\text{do } ([i 0 (+ i 1)]
[j 0 (+ i j)])
[([= i 10) j]
(print j))
\]

Note: Unlike Common Lisp (and “for loops” in many languages), variable is freshly bound
for each iteration. The following example loops 5 times and creates a list of closures, each of
which closes the variable \textit{i}. When you call each closures, you can see that each of them closes
different \textit{i} at the time of the iteration they were created.

\[
(\text{define closures}
(\text{do } ([i 0 (+ i 1)]
[c '() (cons (\text{\textbackslash [] i} c)])
[([= i 5) (\text{\textbackslash reverse} c)]
)))
\]

\[
((\text{car closures)) ⇒ 0
((\text{cadr closures)) ⇒ 1
\]

\textbf{let name ((var init) ...) body ...} \hspace{1cm} \textbf{[Special Form]}

[\textbf{R7RS base}] This variation of \textit{let} is called “named let”. It creates the following procedure
and binds it to \textit{name}, then calls it with \textit{init} ....

\[
(\text{\textbackslash lambda (var ...) body ...})
\]

This syntax itself isn’t necessarily related to iteration. However, the whole point of named
let is that the above lambda expression is within the scope of \textit{name}—that is, you can call
\textit{name} recursively within \textit{body}. Hence this is used very often to write a loop by recursion
(thus, often the procedure is named \textit{loop}, as in the following example.)

\[
(\text{let loop ([x 0] [y '()]}
(if (= x 10)
y
(loop (+ x 1) (cons x y))))
⇒ (9 8 7 6 5 4 3 2 1 0)
\]

Of course you don’t need to loop with a named let; you can call \textit{name} in non-tail position,
pass \textit{name} to other higher-order procedure, etc. Named let exists since it captures a very
common pattern of local recursive procedures. Some Schemers even prefer named let to \textit{do},
for the better flexibility.

The following rewrite rule precisely explains the named let semantics. The tricky use of
\textit{letrec} in the expansion is to make \textit{proc} visible from \textit{body} ... but not from \textit{init} ....

\[
(\text{let proc ((var init) ...) body ...})
≡
((\text{letrec ((proc (\text{\textbackslash lambda (var ...) body ...})})
 Proc)
 \textit{init} ...)
\]
You can use `dotimes` to repeat `body` for a number of times given by `num-expr`, and `dolist` to repeat `body` while traversing a list given by `list-expr`. While `body` is evaluated, `variable` is bound to the current iteration count (in `dotimes`), or the current element in the list (in `dolist`).

\[
\text{\textbf{(dotimes (n 5) (write n))}}
\Rightarrow \text{writes "01234"}
\]

\[
\text{\textbf{(dolist (v '(a b c d e)) (write v))}}
\Rightarrow \text{writes "abcde"}
\]

If you don’t need to refer to `variable`, you can omit it. For example, the following example prints `year!` 10 times:

\[
\text{\textbf{(dotimes (10) (print "yeah!"))}}
\]

If the third element (`result`) is given in `dotimes` or `dolist`, it is evaluated after all repetition is done, and its result becomes the result of `dotimes/dolist`. While `result` is evaluated, `variable` is bound to the number of repetitions (in `dotimes`) or `()` (in `dolist`). It is supported because Common Lisp has it.

Note that a fresh `variable` is bound for each iteration, as opposed to Common Lisp where `variable` is mutated. So if you create a closure closing `variable`, it won’t be overwritten by the subsequent iteration.

If you need more than simple iteration, you can use `do` form, named `let`, or Section 11.11 [Eager comprehensions], page 711, which provides rich way to iterate.

\[
\text{\textbf{while \ expr \ body \ldots}} \quad \text{[Macro]}
\]

\[
\text{\textbf{while \ expr => \ var \ body \ldots}} \quad \text{[Macro]}
\]

\[
\text{\textbf{while \ expr \ guard => \ var \ body \ldots}} \quad \text{[Macro]}
\]

`Var` is an identifier and `guard` is a procedure that takes one argument.

In the first form, `expr` is evaluated, and if it yields a true value, `body` are evaluated. It is repeated while `expr` yields true value.

In the second form, `var` is bound to a result of `expr` in the scope of `body`.

In the third form, the value `expr` yields are passed to `guard`, and the execution of `body` is repeated while `guard` returns a true value. `var` is bound to the result of `expr`.

The return value of `while` form itself isn’t specified.

\[
\text{\textbf{(let ((a '0 1 2 3 4)))}}
\]
\[
\text{\textbf{\hspace{1em} (while (pair? a)}}
\]
\[
\text{\textbf{\hspace{2em} (write (pop! a)))}}
\Rightarrow \text{prints "01234"}
\]

\[
\text{\textbf{(let ((a '0 1 2 3 #f 5 6)))}}
\]
\[
\text{\textbf{\hspace{1em} (while (pop! a)}}
\]
\[
\text{\textbf{\hspace{2em} integer? => var}}
\]
\[
\text{\textbf{\hspace{3em} (write var))}}
\Rightarrow \text{prints "0123"}
\]

\[
\text{\textbf{until \ expr \ body \ldots}} \quad \text{[Macro]}
\]

\[
\text{\textbf{until \ expr \ guard => \ var \ body \ldots}} \quad \text{[Macro]}
\]

Like `while`, but the condition is reversed. That is, the first form repeats evaluation of `expr` and `body` until `expr` yields true. In the second form, the result of `expr` is passed to `guard`, and the execution is repeated until it returns true. `Var` is bound to the result of `expr`. 

(The second form without guard isn’t useful in until, since var would always be bound to #f).

The return value of until form itself isn’t specified.

(let ((a '(0 1 2 3 4)))
  (until (null? a)
    (write (pop! a)))) ⇒ prints "01234"

(until (read-char) eof-object? => ch
  (write-char ch))
⇒ reads from stdin and writes char until EOF is read

4.9 Quasi quotation

quasiquote template

[R7RS base] Quasi quotation is a convenient way to build a structure that has some fixed parts and some variable parts. See the explanation below.

‘template

[R7RS] The syntax ‘x is read as (quasiquote x).

unquote datum ...
unquote-splicing datum ...

[R7RS base] These syntaxes have meaning only when they appear in the template of quasiquoted form. R5RS says nothing about these syntaxes appear outside of quasiquote. Gauche signals an error in such case, for it usually indicates you forget quasiquote somewhere.

R5RS only allows unquote and unquote-splicing to take a single argument; it is undefined if you have (unquote) or (unquote x y) inside quasiquoted form. R6RS allows zero or multi-arguments, and Gauche follows that.

,,datum
,datum

[R7RS] The syntaxes ,x and ,0x are read as (unquote x) and (unquote-splicing x), respectively.

Quasiquote basics

Suppose you want to create a list (foo bar x y), where foo and bar are symbols, and x and y are the value determined at runtime. (For the sake of explanation, let’s assume we have variables x and y that provides those values.) One way to do that is to call the function list explicitly.

(let ((x 0) (y 1))
  (list 'foo 'bar x y)) ⇒ (foo bar 0 1)

You can do the same thing with quasiquote, like this:

(let ((x 0) (y 1))
  '(foo bar ,x ,y)) ⇒ (foo bar 0 1)

The difference between the two notations is that the explicit version quotes the parts that you want to insert literally into the result, while the quasiquote version unquotes the parts that you don’t want to quote.

The quasiquote version gets simpler and more readable when you have lots of static parts with scattered variable parts in your structure.

That’s why quasiquote is frequently used with legacy macros, which are basically a procedure that create program fragments from variable parts provided as macro arguments. See the simple-minded my-if macro that expands to cond form:

(define-macro (my-if test then else)
'(cond (,test ,then)
(else ,else))

(macroexpand '(my-if (< n 0) n (- n)))
⇒ (cond ((< n 0) n) (else (- n)))

Note the two `else`s in the macro definition; one isn’t unquoted, thus appears literally in the output, while another is unquoted and the corresponding macro argument is inserted in its place.

Of course you can use quasiquotes unrelated to macros. It is a general way to construct structures. Some even prefer using quasiquote to explicit construction even most of the structure is variable, for quasiquoted form can be more concise. Gauche also tries to minimize runtime allocation for quasiquoted forms, so it may potentially be more efficient; see "How static are quasiquoted forms?" below.

### Splicing

When `(unquote-splicing expr)` appears in a quasiquoted form, `expr` must evaluate to a list, which is *spliced* into the surrounding context. It’s easier to see examples:

```lisp
(let ((x '(1 2 3)))
 '(a ,@x b))
⇒ (a 1 2 3 b)

(let ((x '(1 2 3)))
 '(a ,x b))
⇒ (a (1 2 3) b)

(let ((x '(1 2 3)))
 '#(a ,@x b))
⇒ #(a 1 2 3 b)
```

Compare the unquote version and unquote-splicing version. Splicing also works within a vector.

#### Multi-argument unquotes

If `unquote` or `unquote-splicing` takes multiple arguments, they are interpreted as if each of its arguments are unquoted or unquote-spliced.

```lisp
;; This is the same result as '((+,(+ 1 2) ,(+ 2 3) ,(+ 3 4))
'((unquote (+ 1 2) (+ 2 3) (+ 3 4)))
⇒ (3 5 7)

;; This is the same result as
;; '((,@(list 1 2) , @(list 2 3) , @(list 3 4)))
'((unquote-splicing (list 1 2) (list 2 3) (list 3 4)))
⇒ (1 2 3 3 4)

;; Edge cases
'((unquote))
⇒ ()
'((unquote-splicing))
⇒ ()
```

It is an error for zero or multiple argument `unquote/unquote-splicing` forms appear which you cannot splice multiple forms into.

```lisp
;; Multiple arguments unquotes are error in non-splicing context
'(unquote 1 2) ⇒ error
'(unquote-splicing 1 2) ⇒ error
```

Note that the abbreviated notations ,x and ,@x are only for single-argument forms. You have to write `unquote` or `unquote-splicing` explicitly for zero or multiple argument forms; thus you don’t usually need to use them. These forms are supported mainly to make the nested unquoting forms such as ,@, @ and ,@ ,@—R5RS cannot handle the case the inner unquote-splicing form expands into zero or multiple forms.
How static are quasiquoted forms?

When quasiquoted form contains variable parts, what happens at runtime is just the same as when an explicit form is used: ‘(,x ,y) is evaluated exactly like (list x y). However, Gauche tries to minimize runtime allocation when a quasiquoted form has static parts.

First of all, if there’s no variable parts in quasiquoted form, like ‘(a b c), the entire form is allocated statically. If there is a static tail in the structure, it is also allocated statically; e.g. ‘((,x a b) (,y c d)) works like (list (cons x '(a b)) (cons y '(c d))).

Furthermore, when an unquoted expression is a constant expression, Gauche embeds it into the static form. If you’ve defined a constant like (define-constant x 3), then the form ‘(,x ,(+ x 1)) is compiled as the constant ‘(3 4). (See Section 4.10 [Definitions], page 68, for the explanation of define-constant form.)

In general it is hard to say which part of quasiquoted form is compiled as a static datum and which part is not, so you shouldn’t write a code that assumes some parts of the structure returned from quasiquote are freshly allocated. In other words, you better avoid mutating such structures.

4.10 Definitions

define variable expression  [Special Form]
define (variable . formals) body ...  [Special Form]
define variable  [R7RS+] This form has different meanings in the toplevel (without no local bindings) or inside a local scope.

On toplevel, it defines a global binding to a symbol variable. In the first form, it globally binds a symbol variable to the value of expression, in the current module.

(define x (+ 1 2))
  x ⇒ 3
(define y (lambda (a) (* a 2)))
  (y 8) ⇒ 16

If variable is already bound in the same module, the subsequent definitions work just like assignments.

(define x 3)
  (define (value-of-x) x)

  (value-of-x x) ⇒ 3

(define x 4)

  (value-of-x x) ⇒ 4

If variable is not bound in the current module, but has an imported bindings, things get interesting but complicated. See Section 4.10.1 [Into the Scheme-Verse], page 72, for the details.

The second form is a syntactic sugar of defining a procedure. It is equivalent to the following form.

(define (name . args) body ...)
  ≡ (define name (lambda args body ...))

The third form is a shorthand of (define variable (undefined)). It is introduced in R6RS (but not a part of R7RS). You can use that form to indicate the initial value doesn’t matter.
If the form appears inside a local scope (internal define), this introduce a local binding of the variable.

Internal defines can appear in the beginning of body of \texttt{lambda} or other forms that introduces local bindings. They are equivalent to a \texttt{letrec*} form, as shown below.

\begin{verbatim}
(lambda (a b)
 (define (cube x) (* x x x))
 (define (square x) (* x x))
 (+ (cube a) (square b)))
≡

(lambda (a b)
  (letrec* ([cube (lambda (x) (* x x x))]
            [square (lambda (x) (* x x))])
     (+ (cube a) (square b))))
\end{verbatim}

Since internal defines are essentially a \texttt{letrec*} form, you can write mutually recursive local functions, and you can use preceding bindings introduced in the same scope to calculate the value to be defined. However, you can’t use a binding that is introduced after an internal define form to calculate its value; if you do so, Gauche may not report an error immediately, but you may get strange errors later on.

\begin{verbatim}
(lambda (a)
 (define x (* a 2))
 (define y (+ x 1)) ; ok to use x to calculate y
 (* a y))
\end{verbatim}

\begin{verbatim}
(lambda (a)
 ;; You can refer to even? in odd?, since the value of even?
 ;; isn’t used at the time odd? is defined; it is only used
 ;; when odd? is called.
 (define (odd? x) (or (= x 1) (not (even? (- x 1)))))
 (define (even? x) (or (= x 0) (not (odd? (- x 1)))))
 (odd? a))
\end{verbatim}

\begin{verbatim}
(lambda (a)
 ;; This is not ok, for defining y needs to use the value
 ;; of x. However, you may not get an error immediately.
 (define y (+ x 1))
 (define x (* a 2))
 (* a y))
\end{verbatim}

Inside the body of binding constructs, internal defines must appear before any expression of the same level. The following code isn’t allowed, for an expression \texttt{(print a)} precedes the \texttt{define} form.

\begin{verbatim}
(lambda (a)
 (print a)
 (define (cube x) (* x x x)) ; error!
 (cube a))
\end{verbatim}

It is also invalid to put no expressions but internal defines inside the body of binding constructs, although Gauche don’t report an error.
Note that `begin` (see Section 4.7 [Grouping], page 62) doesn’t introduce a new scope. Defines in the `begin` act as if `begin` and surrounding parenthesis are not there. Thus these two forms are equivalent.

```
(let ((x 0))
  (begin
    (define (foo y) (+ x y))
    (foo 3))
≡
(let ((x 0))
  (define (foo y) (+ x y))
  (foo 3))
```

```
define-values (var ...) expr
```

```
define-values (var var1 ... var2) expr
```

```
define-values var expr
```

[R7RS base][SRFI-244] `Expr` is evaluated, and each value of the result is bound to each `var`.

In the first form, it is an error unless `expr` yields the same number of values as `vars`.

```
(define-values (lo hi) (min&max 3 -1 15 2))
```

```
lo ⇒ -1
hi ⇒ 15
```

In the second form, `expr` may yield as many values as `var var1 ...` or more; the excess values are made into a list and bound to `var2`.

```
(define-values (a b . c) (values 1 2 3 4))
```

```
a ⇒ 1
b ⇒ 2
c ⇒ (3 4)
```

In the last form, all the values yielded by `expr` are gathered to a list and bound to `var`.

```
(define-values qr (quotient&remainder 23 5))
```

```
qr ⇒ (4 3)
```

You can use `define-values` wherever `define` is allowed; that is, you can mix `define-values` in internal defines.

```
(define (foo . args)
  (define-values (lo hi) (apply min&max args))
  (define len (length args))
  (list len lo hi))
```

```
(foo 1 4 9 3 0 7)
⇒ (6 0 9)
```

```
define-constant variable expression
```

```
define-constant (variable . formals) body ...
```

This form is only effective in toplevel.

Like top-level `define`, it defines a top-level definition of `variable` with the value of `expression`, but additionally tells the compiler that (1) the binding won’t change, and (2) the value of `expression` won’t change from the one computed at the compile time. So the compiler can replace references of `variable` with the compile-time value of `expression`.

An error is signaled when you use `set!` to change the value of `variable`. It is allowed to redefine `variable`, but a warning is printed.
The difference from define-inline below is that the value of expression is computed at the compile time and treated as a literal. Suppose you define x as follows:

\[
\text{(define-constant x (vector 1 2 3))}
\]

Then, the code (list x) is compiled to the same code as (list '#(1 2 3)).

This distinction is especially important when you do AOT (ahead of time) compilation. There's no "internal define-constant", since the compiler can figure out whether a local binding is mutated, and optimize code accordingly, without a help of declarations.

\text{define-inline variable expression} \hspace{1cm} \text{[Special Form]}

\text{define-inline (variable . formals) body ...} \hspace{1cm} \text{[Special Form]}

The second form is a shorthand of (define-inline variable (lambda formals body ...)). If this appears in the position of internal defines, it is the same as internal defines.

If it appears in the toplevel, it defines an inlinable binding. An inlinable binding promises the compiler that the binding won’t change, but unlike constant bindings introduced by define-constant, the actual value of expression may be computed at runtime. Hence the compiler cannot simply replace the references of variable with the compile-time value of expression.

However, if the compiler can determine that the value of expression is to be a procedure, it may inline the procedure where it is invoked.

In the example below, the body of dot3 is inlined where dot3 is called. Furthermore, since the second argument of dot3 is a constant vector, you can see vector-ref on it is computed at compile time (e.g. CONST -1.0 etc.)

\begin{verbatim}
gosh> (define-inline (dot3 a b)
   (+ (* (vector-ref a 0) (vector-ref b 0))
       (* (vector-ref a 1) (vector-ref b 1))
       (* (vector-ref a 2) (vector-ref b 2))))

dot3

dot3
\end{verbatim}
Chapter 4: Core syntax

23 NUMMUL2 ; (* (vector-ref a 2) (vector-ref b 2))
24 NUMADD2 ; (+ (* (vector-ref a 0) (vector-ref b 0))
25 RET

As an extreme case, if both arguments are compile-time constant, dot3 is completely computed at compile time:

gosh> (disasm (^[] (dot3 '#(1 2 3) '#(4 5 6))))
CLOSURE #<closure (#f)>
=== main_code (name=#f, code=0x2a2b8e0, size=2, const=0 stack=0):
signatureInfo: ((#f))
  0 CONSTI(32)
  1 RET

The same inlining behavior may be achieved by making dot3 a macro, but if you use define-inline, dot3 can be used as procedures when needed:

(map dot3 list-of-vectors1 list-of-vectors2)

If dot3 is a macro you can’t pass it as a higher-order procedure.

The inline expansion pass is run top-to-bottom. Inlinable procedure must be defined before used in order to be inlined.

If you redefine an inlinable binding, Gauche warns you, since the redefinition won’t affect already inlined call sites. So it should be used with care—either use it internal to the module, or use it for procedures that won’t change in future. Inlining is effective for performance-critical parts. If a procedure is called sparingly, there’s no point to define it inlinable.

define-in-module module variable expression [Special Form]
define-in-module module (variable . formals) body ... [Special Form]

This form must appear in the toplevel. It creates a global binding of variable in module, which must be either a symbol of the module name or a module object. If module is a symbol, the named module must exist.

Expression is evaluated in the current module.

The second form is merely a syntactic sugar of:

(define-in-module module variable (lambda formals body ...))

Note: to find out if a symbol has definition (global binding) in the current module, you can use global-variable-bound? (see Section 4.13.6 [Module introspection], page 84).

4.10.1 Into the Scheme-Verse

Multiple toplevels are multiple scopes

One upon a time, the Scheme world was simple. We had one single global space we called the toplevel. Toplevel definitions can be understood as side-effects to this global space; if the name hasn’t exist there yet, create a new binding, otherwise, overwrite existing one.

The problem was that it was hard to scale, thus many implementations introduced their own module systems. One of the main agenda of R6RS was to have a module system (which is called “library” in RnRS) consistent with the design of Scheme. Especially, since Scheme’s hygienic macro system captures lexical scope, it is desirable that it interacts with the module system in the same way.

In modern Scheme, “toplevel” of each module creates its own lexical scope, and the definitions are understood in letrec* semantics. Hence, macro systems can consistently treat identifiers as a name associated with a scope.

Suppose you see these toplevel definitions:

(define (odd? n) (if (zero? x) #f (even? (- n 1))))
(define (even? n) (if (zero? x) #t (odd? (- n 1))))

The first appearance of `even?` in the first line is understood as the one defined in the second line. It becomes apparent when we compare it with internal defines:

(let ((even? error))
  (define (odd? n) (if (zero? x) #f (even? (- n 1))))
  (define (even? n) (if (zero? x) #t (odd? (- n 1))))
  ...)

The `even?` in the definition of `odd?` refers to the one defined in the next line, never to the one bound by `let`.

So far, so good.

Now, consider the following toplevel code:

;; Invalid in RnRS, n >= 6
(import (scheme base) (scheme write))
(define orig-error error)
(define (error . args)
  (write args) (newline)
  (apply orig-error args))

The intention is to save the original value of `error`, which is imported from `(scheme base)`, into a variable `orig-error`, then redefine `error` to add logging feature. This technique was popular in pre-R6RS Scheme.

However, with our new toplevel-as-a-scope Scheme, the `error` in `(define orig-error error)` must refer to the one defined in the same scope, which is the new definition below; otherwise lexical scoping gets broken. The value of inner `error` hasn’t been calculated when `orig-error`’s value is calculated, so the above form is an invalid program in terms of RnRS.

In fact, to avoid confusion, R6RS prohibits defining a toplevel variable that conflicts with the imported name (in R7RS the behavior of such program is undefined). In the example above, the name `error` is imported from `(scheme base)` and also defined in the toplevel, hence it’s a violation.

The modern way of such augmentation is to use renaming import:

(import (except (scheme base) error)
  (rename (scheme base) (error r7rs:error))
  (scheme write))
(define (error . args)
  (write args) (newline)
  (apply r7rs:error args))

**Gauche’s take**

Gauche’s module system predates R6RS and R7RS, and it regards a module as a first-class entity and supports class-like inheritance. It is upper-compatible to R7RS libraries, but we take freedom in interpreting R7RS undefined behaviors.

First, you can define toplevel variables that conflict with imported or inherited bindings. The new definition simply shadows the old one.

Second, if multiple toplevel forms are processed at once e.g. it is enclosed in `begin` or the file is read by `include`, we treat them in one scope. That is, if the above `orig-error` example is read by `include`, the first `error` refers to the to-be-defined `error` below. Since the value of `error` hasn’t been calculated by the time it’s used, you’ll get the following error:

*** ERROR: uninitialized variable: error

Third, Gauche compiles and executes each individual toplevel forms (the forms that’s not enclosed in other S-expressions). It is the same as REPL semantics. If each form of `orig-error`
example appears individually on the toplevel, the (define orig-error error) line actually refers to the R7RS error and assign it to orig-error, since we don’t know yet if error will be defined in the same scope.

The third rule is necessary to support REPL semantics, but note that the result would differ when the same file is included. If you can, avoid writing such ambiguous code.

Note: The second behavior is clarified in release 0.9.9 for the better compatibility with R7RS. Before that, the behavior of such case is undefined, but some code might have expected that it works in REPL semantics (the third rule).

In order to support the transition, if you set an environment variable GAUCHE_LEGACY_DEFINE, Gauche treats definitions in the same way as 0.9.8 and before. Note that if you that, you may see Gauche can’t include some valid R7RS code that has multiple libraries in one file.

4.11 Inclusions

include filename ...
include-ci filename ...

[R7RS base] Reads filename ... at compile-time, and insert their contents as if the forms are placed in the includer’s source file, surrounded by begin. The include form reads files as is, while include-ci reads files in case-insensitive way, as if #!fold-case is specified in the beginning of the file (see Section 2.4 [Case-sensitivity], page 15).

The coding magic comment in each file is honored while reading that file (see Section 2.3 [Multibyte scripts], page 14).

If filename is absolute, the file is just searched. If it is relative, the file is first searched relative to the file containing the include form, then the directories in *load-path* are tried.

Example: Suppose a file a.scm contains the following code:

(define x 0)
(define y 1)

You can include this file into another source, like this:

(define (foo)
  (include "a.scm")
  (list x y))

It works as if the source is written as follows:

(define (foo)
  (begin
    (define x 0)
    (define y 1))
  (list x y))

(Note: In version 0.9.4, include behaved differently when pathname begins with either ./ or ../—in which case the file is searched relative to the current working directory of the compiler. It is rather an artifact of include sharing file search routine with load. But unlike load, which is a run-time operation, using relative path to the current directory won’t make much sense for include, so we changed the behavior in 0.9.5.)

Gauche has other means to incorporate source code from another files. Here’s the comparison.

require (use and extend calls require internally)
- Both require and include work at compile-time.
- Require works only in toplevel context, while include can be anywhere.
- Require reads the file only once (second and later require on the same file becomes no-op), while include reads the file every place it appears.
• The file is searched from \texttt{*load-path*}. The location of the file \texttt{require} form appears doesn’t matter. (You can add directories relative to the requiring file using the \texttt{:relative} flag in \texttt{add-load-path}, though).

• Even if the current module is changed by \texttt{select-module} inside the required file, it is only effective while the required file is read. On the other hand, \texttt{include} inserts any S-expressions in the included file to the place \texttt{include} appears, so the effect of \texttt{select-module} persists after \texttt{include} form (Note: Encoding magic comment and \texttt{#!/fold-case/#!/no-fold-case} are dealt with by the reader, so those effect is contained in the file even with \texttt{include}).

• It is forbidden to the file loaded by \texttt{require} to insert a toplevel binding without specifying a module. In other words, the file you require should generally use \texttt{define-module}, \texttt{select-module} or \texttt{define-library}. See Section 6.22.3 [Require and provide], page 288, for further discussion. On the other hand, \texttt{include} has no such restrictions.

\texttt{load}

• Works at runtime, while \texttt{include} works at compile-time.

• Works only in toplevel context, while \texttt{include} can be anywhere.

• The file is searched from \texttt{*load-path*}, except when the file begins with ./ or ../, in which case it is first tried relative to the current directory before being searched from \texttt{*load-path*}.

• As the case with \texttt{require}, change of the current module won’t persist after \texttt{load}.

Usually, \texttt{require} (or \texttt{use} and \texttt{extend}) are better way to incorporate sources in other files. The \texttt{include} form is mainly for the tricks that can’t be achieved with \texttt{require}. For example, you have a third-party R5RS code and you want to wrap it with Gauche module system. Using \texttt{include}, you place the following small source file along the third-party code, and you can load the code with \texttt{(use third-party-module)} without changing the original code at all.

\begin{verbatim}
(define-module third-party-module
 (export proc ...)
 (include "third-party-source.scm")
)
\end{verbatim}

4.12 Feature conditional

The \texttt{cond-expand} macro

Sometimes you need to have a different piece of code depending on available features provided by the implementation and/or platform. For example, you may want to switch behavior depending on whether networking is available, or to embed an implementation specific procedures in otherwise-portable code.

In C, you use preprocessor directives such as \texttt{#ifdef}. In Common Lisp, you use reader macro \texttt{#+} and \texttt{#-}. In Scheme, you have \texttt{cond-expand}:

\begin{verbatim}
cond-expand (feature-requirement command-or-definition ...) . . . [Macro]
[R7RS base] This macro expands to command-or-definition ... if feature-requirement is supported by the current platform.

feature-requirement must be in the following syntax:

feature-requirement
  : feature-identifier
  | (and feature-requirement ...)
\end{verbatim}
| (or  feature-requirement ...)
| (not feature-requirement)
| (library library-name)

The macro tests each feature-requirement in order, and if one is satisfied, the macro itself expands to the corresponding command-or-definition . . .

The last clause may have else in the position of feature-requirement, to make the clause expanded if none of the previous feature requirement is fulfilled.

If there's neither a satisfying clause nor else clause, cond-expand form throws an error. It is to detect the case early that the platform doesn't have required features. If the feature you're testing is optional, that is, your program works without the feature as well, add empty else clause as follows.

(cond-expand
   [feature expr] ; some optional feature
   [else])

feature-identifier is a symbol that indicates a feature. If such a feature is supported in the current platform, it satisfies the feature-requirement. You can do boolean combination of feature-requirements to compose more complex conditions.

The form (library library-name) is added in R7RS, and it is fulfilled when the named library is available. For the library-name, you can use both R7RS library name (e.g. (srfi 13)) and traditional Gauche module name (e.g. srfi.13).

Here's a typical example: Suppose you want to have implementation-specific part for Gauche, Chicken Scheme and ChibiScheme. Most modern Scheme implementations defines a feature-identifier to identify itself. You can write the conditional part as follows:

```
(cond-expand
  [gauche  (gauche-specific-code)]
  [(or chicken chibi) (chicken-chibi-specific-code)]
  [else   (fallback-code)]
)
```

It is important that the conditions of cond-expand is purely examined at the macro-expansion time, and unfulfilled clauses are discarded. Thus, for example, you can include macro calls or language extensions that may not be recognized on some implementations. You can also conditionally define global bindings.

Compare that to cond, which examines conditions at runtime. If you include unsupported macro call in one of the conditions, it may raise an error at macro expansion time, even if that clause will never be executed on the platform. Also, it is not possible to conditionally define global bindings using cond.

There's a caveat, though. Suppose you want to save the result of macro expansion, and run the expanded result later on other platforms. The result code is based on the features of the platform the macro expansion takes place, which may not agree with the features of the platform the code will run. (This issue always arises in cross-compiling situation in general.)

NB: When cond-expand was defined as SRFI-0, there was no standard way to name Scheme libraries. So SRFI-0 suggested to use srfi-N as feature identifier to check the support of a specific srfi. We deprecate this usage. It should be replaced with (library srfi.N). See Section 11.1 [Testing availability of SRFIs], page 689, for the details.

As of 0.9.13, a warning is issued when srfi-N feature identifier is used in cond-expand if the environment variable GAUCHE_WARN_SRFI_FEATURE_ID is set. In future versions the warning will be default; we recommend to set this environment variable to detect code to update.

See below for the list of feature identifiers defined in Gauche.
Gauche-specific feature identifiers

\texttt{gauche}
\texttt{gauche-X.X.X}

Indicates you’re running on Gauche. It is useful to put Gauche-specific code in a portable program. \texttt{X.X.X} is the gauche’s version (e.g. \texttt{gauche-0.9.4}), in case you want to have code for specific Gauche version. (Such feature identifier is suggested by R7RS; but it might not be useful if we don’t have means to compare versions. Something to consider in future versions.)

\texttt{gauche.os.windows}
\texttt{gauche.os.cygwin}

Defined on Windows-native platform and Cygwin/Windows platform, respectively. If neither is defined you can assume it’s a unix variant. (Cygwin is supposedly unix variant, but corners are different enough to deserve it’s own feature identifier.)

\texttt{gauche.ces.utf8}
\texttt{gauche.ces.eucjp}
\texttt{gauche.ces.sjis}
\texttt{gauche.ces.none}

Either one of these is defined based on Gauche’s native character encoding scheme. See Section 2.2 [Multibyte strings], page 13, for the details.

\texttt{gauche.net.tls}
\texttt{gauche.net.tls.axtls}
\texttt{gauche.net.tls.mbedtls}

Defined if the runtime supports TLS in networking. The two sub feature identifiers, \texttt{gauche.net.tls.axtls} and \texttt{gauche.net.tls.mbedtls}, are defined if each subsystem \texttt{axTLS} and \texttt{mbedTLS} is supported, respectively.

\texttt{gauche.net.ipv6}

Defined if the runtime supports IPv6. Note that this only indicates Gauche has been built with IPv6 support; the OS may not allow IPv6 features, in that case you’ll get system error when you try to use IPv6.

\texttt{gauche.sys.threads}
\texttt{gauche.sys.pthreads}
\texttt{gauche.sys.wthreads}

If the runtime supports multithreading, \texttt{gauche.sys.threads} is defined (see Section 9.35 [Threads], page 530). Multithreading is based on either POSIX pthreads or Windows threads. The former defines \texttt{gauche.sys.pthreads}, and the latter defines \texttt{gauche.sys.wthreads}.
Those are defined based on the availability of these system features of the platform.

**R7RS feature identifiers**

- **r7rs**: Indicates the implementation complies r7rs.
- **exact-closed**: Exact arithmetic operations are closed; that is, dividing an exact number by a non-zero exact number always yields an exact number.
- **ieee-float**: Using IEEE floating-point number internally.
- **full-unicode**: Full unicode support.
- **ratios**: Rational number support
- **posix**
- **windows**: Either one is defined, according to the platform.
- **big-endian**
- **little-endian**: Either one is defined, according to the platform.

### 4.13 Modules

This section describes the semantics of Gauche modules and its API. See also Section 3.7 [Writing Gauche modules], page 38, for the conventions Gauche is using for its modules.

For R7RS programs, they are called “libraries” and have different syntax than Gauche modules. See Section 10.2.1 [R7RS library form], page 584, for the details.

#### 4.13.1 Module semantics

Module is an object that maps symbols onto **bindings**, and affects the resolution of global variable reference.

Unlike CommonLisp’s packages, which map names to symbols, in Gauche symbols are **eq?** in principle if two have the same name (except uninterned symbols; see Section 6.7 [Symbols], page 155). However, Gauche’s symbol doesn’t have a ’value’ slot in it. From a given symbol, a
A module finds its binding that keeps a value. Different modules can associate different bindings to the same symbol, that yield different values.

;; Makes two modules A and B, and defines a global variable 'x' in them
(define-module A (define x 3))
(define-module B (define x 4))

;; #<symbol 'x'> ---[module A]--> #<binding that has 3>
(with-module A x) ⇒ 3

;; #<symbol 'x'> ---[module B]--> #<binding that has 4>
(with-module B x) ⇒ 4

A module can export a part or all of its bindings for other module to use. A module can import other modules, and their exported bindings become visible to the module. A module can import any number of modules.

(define-module A
  (export pi)
  (define pi 3.1416))

(define-module B
  (export e)
  (define e 2.71828))

(define-module C
  (import A B))

(select-module C)
(* pi e) ⇒ 8.539748448

A module can also be inherited, that is, you can extend the existing module by inheriting it and adding new bindings and exports. From the new module, all ancestor’s bindings (including non-exported bindings) are visible. (A new module inherits the gauche module by default, which is why the built-in procedures and syntax of gauche are available in the new module). From outside, the new module looks like having all exported bindings of the original module plus the newly defined and exported bindings.

;; Module A defines and exports deg->rad.
;; A binding of pi is not exported.
(define-module A
  (export deg->rad)
  (define pi 3.1416) ;; not exported
  (define (deg->rad deg) (* deg (/ pi 180))))

;; Module Aprime defines and exports rad->deg.
;; The binding of pi is visible from inside Aprime.
(define-module Aprime
  (extend A)
  (export rad->deg)
  (define (rad->deg rad) (* rad (/ 180 pi))))

;; Module C imports Aprime.
(define-module C
  (import Aprime)
  ;; Here, both deg->rad and rad->deg are visible,
At any moment of the compilation, there is one "current module" available, and the global variable reference is looked for from the module. If there is a visible binding of the variable, the variable reference is compiled to the access of the binding. If the compiler can’t find a visible binding, it marks the variable reference with the current module, and delays the resolution of binding at the time the variable is actually used. That is, when the variable is referenced at run time, the binding is again looked for from the marked module (not the current module at the run time) and if found, the variable reference code is replaced for the the code to access the binding. If the variable reference is not found even at run time, an 'undefined variable’ error is signaled.

Once the appropriate binding is found for the global variable, the access to the binding is hard-wired in the compiled code and the global variable resolution will never take place again.

The definition special form such as define and define-syntax inserts the binding to the current module. Thus it may shadow the binding of imported or inherited modules.

The resolution of binding of a global variable happens like this. First, the current module is searched. Then, each imported module is taken in the reverse order of import, and searched, including each module’s ancestors. Note that import is not transitive; imported module list is not chased recursively. Finally, ancestors of the current module are searched in order.

This order is important when more than one modules defines the same name and your module imports both. Assuming your module don’t define that name, if you first import a module A then a module B, you’ll see B’s binding.

If you import A, then B, then A again, the last import takes precedence; that is, you’ll see A’s binding.

If two modules you want to use exports bindings of the same name and you want to access both, you can add prefix to either one (or both). See Section 4.13.4 [Using modules], page 81, for the details.

4.13.2 Modules and libraries

Modules are run-time data structure; you can procedurally create modules with arbitrary names at run-time.

However, most libraries use modules to create their own namespace, so that they can control which bindings to be visible from library users. (This “library” is a general term, broader than R7RS “library”)

Usually a library is provided in the form of one or more Scheme source file(s), so it is convenient to have a convention to map module names to file names, and vice versa; then, you can load a library file and import its module by one action with use macro, for example.

For the time being, Gauche uses a simple rules for this mapping: Module names are organized hierarchically, using period '.' for separator, e.g. gauche.mop.validator. If such a module is requested and doesn’t exist in the current running environment, Gauche maps the module name to a pathname by replacing periods to directory separator, i.e. gauche/mop/validator, and look for gauche/mop/validator.scm in the load paths.

Note that this is just a default behavior. Theoretically, one Scheme source file may contain multiple modules, or one module implementation may span to multiple files. In future, there may be some hook to customize this mapping for special cases. So, when you are writing routines that deal with modules and library files, do not apply the above default rule blindly. Gauche provides two procedures, module-name->path and path->module-name, to do mapping for you (see Section 4.13.6 [Module introspection], page 84, for details).
4.13.3 Defining and selecting modules

**define-module name body ...**  
[Special Form]
Name must be a symbol. If a module named name does not exist, create one. Then evaluates body sequentially in the module.

**select-module name**  
[Special Form]
Makes a module named name as the current module. It is an error if no module named name exists.
If select-module is used in the Scheme file, its effect is limited inside the file, i.e. even if you load/require a file that uses select-module internally, the current module of requirer is not affected.

**with-module name body ...**  
[Special Form]
Evaluates body sequentially in the module named name. Returns the last result(s). If no module named name, an error is signaled.

**current-module**  
[Special Form]
Evaluates to the current module in the compile context. Note that this is a special form, not a function. Module in Gauche is statically determined at compile time.

```scheme
(define-module foo
 (export get-current-module)
 (define (get-current-module) (module-name (current-module))))

(define-module bar
 (import foo)
 (get-current-module)) ⇒ foo ; not bar
```

4.13.4 Using modules

**export spec ...**  
[R7RS base] Makes bindings specified by each spec available to modules that imports the current module.
Each spec can be either one of the following forms, where name and exported-name are symbols.

- `name` The binding with name is exported.
- `(rename name exported-name)` The binding with name is exported under an alias exported-name.

Note: In Gauche, export is just a special form you can put in the middle of the program, whereas R7RS defines export as a library declaration, that can only appear immediately below define-library form. See Section 10.2.1 [R7RS library form], page 584, for the details.

**export-all**  
[Special Form]
Makes all bindings in the current module available to modules that imports it.

**import import-spec ...**  
[Special Form]
Makes all or some exported bindings in the module specified by import-spec available in the current module. The syntax of import-spec is as follows.

```
<import-spec> : <module-name>
 | ( <module-name> <import-option> ...)
```
<import-option> : :only (<symbol> ...)
| :except (<symbol> ...)
| :rename (((<symbol> <symbol>) ...)
| :prefix <symbol>

<module-name> : <symbol>
The module named by module-name should exist when the compiler sees this special form.

Imports are not transitive. The modules that module-names are importing are not automatically imported to the current module. This keeps modules' modularity; a library module can import whatever modules it needs without worrying about polluting the namespace of the user of the module.

import-option can be used to change how the bindings are imported. With :only, only the bindings with the names listed in <symbol> ... are imported. With :except, the exported bindings except the ones with the listed names are imported. With :rename, the binding of each name in the first of two-symbol list is renamed to the second of it. With :prefix, the exported bindings are visible with the names that are prefixed by the symbol to the original names. Without import options, all the exported bindings are imported without a prefix.

(define-module M (export x y)
  (define x 1)
  (define y 2)
  (define z 3))

(import M)

x ⇒ 1
z ⇒ error. z is not exported from M

(import (M :only (y)))

x ⇒ error. x is not in :only list.

(import (M :except (y)))

y ⇒ error. y is excluded by :except.

(import (M :prefix M:))

x ⇒ error
M:x ⇒ 1
M:y ⇒ 2

If more than one import option are given, it is processed as the order of appearance. That is, if :prefix comes first, then :only or :except has to list the name with prefix.

Note: R7RS has import form, which has slightly different syntax and semantics. See Section 10.1.2 [Three forms of import], page 582, for the details.

use name :key only except rename prefix

A convenience macro that combines module imports and on-demand file loading. Basically, (use foo) is equivalent to the following two forms:

(require "foo")
(import foo)
That is, it loads the library file named "foo" (if not yet loaded) which defines a module named foo in it, and then import the module foo into the current module.

The keyword argument only, except, and prefix are passed to import as the import options.

```
(use srfi.1 :only (iota) :prefix srfi-1:)
```

```
(srfi-1:iota 3) ⇒ (0 1 2)
```

Although the files and modules are orthogonal concept, it is practically convenient to separate files by modules. Gauche doesn't force you to do so, and you can always use require and import separately. However, all modules provided with Gauche are arranged so that they can be used by use macro.

If a module is too big to fit in one file, you can split them into several subfiles and one main file. The main file defines the module, and either loads, requires, or autoloads subfiles.

Actually, the file pathname of the given module name is obtained by the procedure module-name->path below. The default rule is to replace periods '.' in the name for '/'; for example, (use foo.bar.baz) is expanded to:

```
(require "foo/bar/baz")
(import foo.bar.baz)
```

This is not very Scheme-ish way, but nevertheless convenient. In future, there may be some mechanism to customize this mapping.

The file to be used must have explicit module selection to have any toplevel definitions (usually via define-module/select-module pair or define-library). If you get an error saying “Attempted to create a binding in a sealed module: module: #<module gauche.require-base>”, that’s because the file lacks module selection. See Section 6.22.3 [Require and provide], page 288, for further discussion.

### 4.13.5 Module inheritance

The export-import mechanism doesn’t work well in some cases, such as:

- You want to create a module that is mostly the same as the existing one, but adding or altering some definitions.
- You wrote a bunch of related modules that are often used together, and not want your users to repeat a bunch of 'use' forms every time they use your module.

You can use module inheritance in these cases.

```
extend module-name ...
```

[Macro]

Makes the current module inherit from named modules. The current inheritance information is altered by the inheritance information calculated from given modules.

A new module inherits from gauche module when created. If you put (extend scheme) in that module, for example, the module resets to inherit directly from scheme module that has only bindings defined in R5RS, hence, after the extend form, you can’t use 'import' or any other gauche-specific bindings in the module.

If a named module is not defined yet, extend tries to load it, using the same convention use macro does.

A module can inherit multiple modules, exactly the same way as a class can inherit from multiple classes. The resolution of order of inheritance needs to be explained a bit.

Each module has a module precedence list, which lists modules in the order of how they are searched. When the module inherits multiple modules, module precedence lists of inherited modules are merged into a single list, keeping the constraints that: (1) if a module A appears before module B in some module precedence list, A has to appear before B in the resulting
module precedence list; and (2) if a module A appears before module B in `extend` form, A has to appear before B in the resulting module precedence list. If no precedence list can be constructed with these constraints, an error is signaled.

For example, suppose you wrote a library in modules `mylib.base`, `mylib.util` and `mylib.system`. You can bundle those modules into one module by creating a module `mylib`, as follows:

```
(define-module mylib
  (extend mylib.system mylib.util mylib.base))
```

The user of your module just says `(use mylib)` and all exported symbols from three sub-modules become available.

### 4.13.6 Module introspection

This subsection lists procedures that operates on modules at run-time. With these procedures you can introspect the modules, create new modules procedurally, or check the existence of certain modules/libraries, for example. However, don’t forget that modules are primarily compile-time structures. Tweaking modules at run-time is only for those who know what they are doing.

```scheme
<module>
A module class.
```

```scheme
module? obj
Returns true if obj is a module.
```

```scheme
find-module name
Returns a module object whose name is a symbol name. If the named module doesn’t exist, #f is returned.
```

```scheme
make-module name :key if-exists
Creates and returns a module that has symbol name. If the named module already exists, the behavior is specified by if-exists keyword argument. If it is :error (default), an error is signaled. If it is #f, #f is returned.

Note that creating modules on-the-fly isn’t usually necessary for ordinal scripts, since executing already written program requires modules to be specified by name, i.e. syntax `define-module, import, extend, with-module` all take module names, not module objects. It is because module are inherently compile-time structures. However, there are some cases that dynamically created modules are useful, especially when the program itself is dynamically created. You can pass a module to `eval` to compile and evaluate such dynamically created programs in it (see Section 6.20 [Eval and repl], page 258).

You can also pass #f to name to create anonymous module. Anonymous modules can’t be looked up by `find-module`, nor can be imported or inherited (since `import` and `extend` take module names, not modules). It is useful when you want to have a temporary, segregated namespace dynamically—for example, you can create an anonymous module to evaluate code fragments sent from other program, and discards the module when the connection is terminated. Anonymous modules are not registered in the system dictionary and are garbage collected when nobody keeps reference to it.

R7RS provides another way to create a transient module with `environment` procedure. see Section 10.2.7 [R7RS eval], page 588, for the details.

```scheme
all-modules
Returns a list of all named modules. Anonymous modules are not included.
```
module-name module  [Function]
module-imports module  [Function]
module-exports module  [Function]
module-table module  [Function]

Accessors of a module object. Returns the name of the module (a symbol), list of imported modules, list of exported symbols, and a hash table that maps symbols to bindings, of the module are returned, respectively.

It is an error to pass a non-module object.

module-parents module  [Function]
module-precedence-list module  [Function]

Returns the information of module inheritance. Module-parents returns the modules module directly inherits from. Module-precedence-list returns the module precedence list of module (see Section 4.13.5 [Module inheritance], page 83).

global-variable-bound? module symbol  [Function]

Returns true if symbol’s global binding is visible from module. Module must be a module object or a symbol name of an existing module.

Note: there used to be the symbol-bound? procedure to check whether a global variable is bound. It is deprecated and the new code should use global-variable-bound? instead. The reason of change is that because of the name symbol-bound? and the fact that it assumes current-module by default, it gives an illusion as if a global bound value is somewhat ‘stored’ in a symbol itself (like CommonLisp’s model). It caused a lot of confusion when the current module differs between compile-time and runtime. The new name and API made it clear that you are querying module’s property.

global-variable-ref module symbol :optional default  [Function]

Returns a value globally bound to the symbol visible from module. Module must be a module object or a symbol name of an existing module. If there’s no visible global binding from module for symbol, an error is signaled, unless the default argument is provided, in which case it is returned instead.

module-name->path symbol  [Function]

Converts a module name symbol to a fragment of pathname string (which you use for require and provide).

path->module-name string  [Function]

Reverse function of module-name->path.

If you want to find out specific libraries and/or modules are installed in the system and available from the program, see Section 6.22.5 [Operations on libraries], page 289.

4.13.7 Predefined modules

Several modules are predefined in Gauche.

null  [Builtin Module]

This module corresponds to the null environment referred in R5RS. This module contains only syntactic bindings of R5RS syntax.

scheme  [Builtin Module]

This module contains all the binding of null module, and the binding of procedures defined in R5RS.
Note that if you change the current module to null or scheme by select-module, there will be no way to switch back to other modules, since module-related syntaxes and procedures are not visible from null and scheme modules.

`gauche`  
[Builtin Module]  
This module contains all the bindings of scheme module, plus Gauche specific built-in procedures.

`user`  
[Builtin Module]  
This module is the default module the user code is compiled. all the bindings of `gauche` module is imported.

`gauche.keyword`  
[Builtin Module]  
When Gauche is running with GAUCHE_KEYWORD_IS_SYMBOL mode (default) keywords (symbols beginning with :) is automatically bound to itself in these modules. (see Section 6.8 [Keywords], page 157, for the details.) The `keyword` module doesn’t export those bindings, while `gauche.keyword` does. The former is intended to be used internally; the programmer need to know the latter.

If you use the default module inheritance, you don’t need to use this module, since the `keyword` module is included in the inheritance chain. If you don’t inherit `gauche` module, however, importing the `gauche.keyword` module gives you access to the keywords without quotes. For example, R7RS programs and libraries would require either `(import (gauche keyword))` or `(import (gauche base))` (the latter inherits `gauche.keyword`), or you have to quote all keywords.

The following R7RS program imports `gauche.base`; it makes gauche built-in identifiers, and all self-bound keywords, available:

```
;; R7RS program
(import (scheme base)
(gauche base)) ; import gauche builtins and keywords

;; You can use :directory without quote, for it is bound to itself.
(sys-exec "ls" '("ls" "-l") :directory ")")
```

If you use more sophisticated import tricks, however, keep in mind that keywords are just imported symbols by default. The following code imports Gauche builtin identifiers with prefix `gauche/`. That causes keywords, imported via inheritance, also get the same prefix; if you don’t want to bother adding prefix to all keywords or quote them, import `gauche.keyword` separately.

```
;; R7RS program
(import (scheme base)
(prefix (gauche base) gauche/) ; use gauche builtin with gauche/ prefix
(gauche keyword)) ; imports keywords

;; Without importing gauche.keyword,
;; you need to write ':directory
(gauche/sys-exec "ls" '("ls" "-l") :directory ")")
```

`gauche.require-base`  
[Builtin Module]  
This is a module when use or require load the file. This is not meant to be used directly; just to prevent library files lacking module switching from causing unexpected side effects. See Section 4.13.4 [Using modules], page 81, for more details.
5 Macros

Macro of Lisp-family language is very different feature from ones of other languages, such as C preprocessor macros. It allows you to extend the original language syntax. You can use macros to change Gauche syntax so that you can run a Scheme program written to other Scheme implementations, and you can even design your own mini-language to solve your problem easily.

Gauche supports hygienic macros, which allows to write safe macros by avoiding name collisions. If you know traditional Lisp macros but new to hygienic macros, they might seem confusing at first. We have an introductory section (Section 5.1 [Why hygienic?], page 87) for those who are not familiar with hygienic macros; if you know what they are, you can skip the section.

5.1 Why hygienic?

Lisp macro is a programmatic transformation of source code. A macro transformer is a procedure that takes a subtree of source code, and returns a reconstructed tree of source code.

The traditional Lisp macros take the input source code as an S-expression, and returns the output as another S-expression. Gauche supports that type of macro, too, with define-macro form. Here’s the simple definition of when with the traditional macro.

```
(define-macro (when test . body)
  '(if ,test (begin ,@body)))
```

For example, if the macro is used as `(when (zero? x) (print "zero") 'zero), the above macro transformer rewrites it to `(if (zero? x) (begin (print "zero") 'zero)). So far, so good.

But what if the when macro is used in an environment where the names begin or if is bound to nonstandard values?

```
(let ([begin list])
  (when (zero? x) (print "zero") 'zero))
```

The expanded result would be as follows:

```
(let ([begin list])
  (if (zero? x) (begin (print "zero") 'zero)))
```

This obviously won’t work as the macro writer intended, since begin in the expanded code refers to the locally bound name.

This is a form of variable capture. Note that, when Lisp people talk about variable capture of macros, it often means another form of capture, where the temporary variables inserted by a macro would unintentionally capture the variables passed to the macro. That kind of variable capture can be avoided easily by naming the temporary variables something that never conflict, using gensym.

On the other hand, the kind of variable capture in the above example can’t be avoided by gensym, because `(let ([begin list] ...) part isn’t under macro writer’s control. As a macro writer, you can do nothing to prevent the conflict, just hoping the macro user won’t do such a thing. Sure, rebinding begin is a crazy idea that nobody perhaps wants to do, but it can happen on any global variable, even the ones you define for your library.

Various Lisp dialects have tried to address this issue in different ways. Common Lisp somewhat relies on the common sense of the programmer—you can use separate packages to reduce the chance of accidental conflict but can’t prevent the user from binding the name in the same package. (The Common Lisp spec says it is undefined if you locally rebind names of CL standard symbols; but it doesn’t prevent you from locally rebinding symbols that are provided by user libraries.)
Clojure introduced a way to directly refer to the toplevel variables by a namespace prefix, so it can bypass whatever local bindings of the same name (also, it has a sophisticated quasiquote form that automatically renames free variables to refer to the toplevel ones). It works, as far as there are no local macros. With local macros, you need a way to distinguish different local bindings of the same name, as we see in the later examples. Clojure’s way can only distinguish between local and toplevel bindings. It’s ok for Clojure which doesn’t have local macros, but in Scheme, we prefer uniform and orthogonal axioms—if functions can be defined locally with lexical scope, why not macros?

Let’s look at the local macro with lexical scope. For the sake of explanation, suppose we have hypothetical local macro binding form, let-macro, that binds a local identifiers to a macro transformer. (We don’t actually have let-macro; what we have is let-syntax and letrec-syntax, which have slightly different way to call macro transformers. But here let-macro may be easier to understand as it is similar to define-macro.)

```
(let ([f (^x (* x x))])
  (let-macro ([m (^
[expr1 expr2] `(+(f ,expr1) (f ,expr2)))]
    (let ([f (^x (+ x x))])
      (m 3 4)))); [1]
```

The local identifier `m` is bound to a macro transformer that takes two expressions, and returns an S-expression. So, the (m 3 4) form [1] would be expanded into (+ (f 3) (f 4)). Let’s rewrite the above expression with the expanded form. (After expansion, we no longer need let-macro form, so we don’t include it.)

```
(let ([f (^x (* x x))])
  (let ([f (^x (+ x x))])
    (+ (f 3) (f 4)))); [2]
```

Now, the question. Which binding `f` in the expanded form [2] should refer? If we literally interpret the expansion, it would refer to the inner binding (^x (+ x x)). However, following the Scheme’s scoping principle, the outer code should be fully understood regardless of inner code:

```
(let ([f (^x (* x x))])
  (let-macro ([m (^
[expr1 expr2] `(+(f ,expr1) (f ,expr2)))]
    ;; The code here isn’t expected to accidentally alter
    ;; the behavior defined outside.
    ))
```

The macro writer may not know the inner let shadows the binding of `f` (the inner forms may be included, or may be changed by other person who didn’t fully realize the macro expansion needs to refer outer `f`).

To ensure the local macro to work regardless of what’s placed inside let-macro, we need a sure way to refer the outer `f` in the result of macro expansion. The basic idea is to “mark” the names inserted by the macro transformer `m`—which are `f` and `+`—so that we can distinguish two `f`’s.

For example, if we would rewrite the entire form and renames corresponding local identifiers as follows:

```
(let ([f_1 (^x (* x x))])
  (let-macro ([m (^
[expr1 expr2] `(+(f_1 ,expr1) (f_1 ,expr2)))]
    ;; The code here isn’t expected to accidentally alter
    ;; the behavior defined outside.
    ))
```

Then the naive expansion would correctly preserve scopes; that is, expansion of `m` refers `f_1`, which wouldn’t conflict with inner name `f_2`:

```
(let ([f_1 (^x (* x x))])
```
(let ([f_2 (^x (+ x x))]
    (+ (f_1 3) (f_1 4))))

(You may notice that this is similar to lambda calculus treating lexical bindings with higher
order functions.)

The above example deal with avoiding \( f \) referred from the macro definition (which is, in fact, \( f_1 \)) from being shadowed by the binding of \( f \) at the macro use (which is \( f_2 \)).

Another type of variable capture (the one most often talked about, and can be avoided by gensym) is that a variable in macro use site is shadowed by the binding introduced by a macro definition. We can apply the same renaming strategy to avoid that type of capture, too. Let’s see the following example:

(let ([f (^x (* x x))])
  (let-macro ([m (^[expr1] '(let ([f (^x (+ x x))]) (f ,expr1))))]
    (m (f 3))))

The local macro inserts binding of \( f \) into the expansion. The macro use \((m (f 3))\) also
contains a reference to \( f \), which should be the outer \( f \), since the macro use is lexically outside
of the \( let \) inserted by the macro.

We could rename \( f \)’s according to its lexical scope:

(let ([f_1 (^x (* x x))])
  (let-macro ([m (^[expr1] '(let ([f_2 (^x (+ x x))]) (f_2 ,expr1))))]
    (m (f_1 3))))

Then expansion unambiguously distinguish two \( f \)’s.

(let ([f_1 (^x (* x x))])
  (let ([f_2 (^x (+ x x))])
    (f_2 (f_1 3))))

This is, in principle, what hygienic macro is about (well, almost). In reality, we don’t rename
everything in batch. One caveat is in the latter example—we statically renamed \( f \) to \( f_2 \), but
it is possible that the macro recursively calls itself, and we have to distinguish \( f \)’s introduced in
every individual expansion of \( m \). So macro expansion and renaming should work together.

There are multiple strategies to implement it, and the Scheme standard doesn’t want to bind
implementations to single specific strategy. The standard only states the properties the macro
system should satisfy, in two concise sentences:

- If a macro transformer inserts a binding for an identifier (variable or keyword), the
  identifier will in effect be renamed throughout its scope to avoid conflicts with other
  identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers
to the binding that was visible where the transformer was specified, regardless of
any local bindings that surround the use of the macro.

Just from reading this, it may not be obvious how to realize those properties, and the existing
hygienic macro mechanisms (e.g. syntax-rules) hide the “how” part. That’s probably one of
the reason some people feel hygienic macros are difficult to grasp. It’s like continuations—its
description is concise but at first you have no idea how it works; then, through experience, you
become familiarized yourself to it, and then you reread the original description and understand
it says exactly what it is.

This introduction may not answer how the hygienic macro realizes those properties, but I hope
it showed what it does and why it is needed. In the following chapters we introduce a couple
of hygienic macro mechanisms Gauche supports, with examples, so that you can familiarize
yourself to the concept.
5.2 Hygienic macros

Macro bindings
The following forms establish bindings of name and a macro transformer created by transformer-spec. The binding introduced by these forms shadows a binding of name established in outer scope, if there’s any.

For toplevel bindings, it will shadow bindings of name imported or inherited from other modules (see Section 4.13 [Modules], page 78). (Note: This toplevel shadowing behavior is Gauche’s extension; in R7RS, you shouldn’t redefine imported bindings, so the portable code should avoid it.)

The effect is undefined if you bind the same name more than once in the same scope.

The transformer-spec can be either one of syntax-rules form, er-macro-transformer form, or another macro keyword or syntactic keyword. We’ll explain them later.

define-syntax name transformer-spec

[Special Form]
[R7RS base] If this form appears in toplevel, it binds toplevel name to a macro transformer defined by transformer-spec.

If this form appears in the declaration part of body of lambda (internal define-syntax), let and other similar forms, it binds name locally within that body. Conceptually, internal define-syntaxes on the same level are treated like letrec-syntax. However, mere appearance of define-syntax does not create another scope; for example, you can interleave internal define and internal define-syntax within the same scope. It is important, though, that the local macros defined by internal define-syntax should not be required to expand macro uses before the definition.

let-syntax ((name transformer-spec) ...) body

[Special Form]

letrec-syntax ((name transformer-spec) ...) body

[Special Form]

[R7RS base] Defines local macros. Each name is bound to a macro transformer as specified by the corresponding transformer-spec, then body is expanded. With let-syntax, transformer-spec is evaluated with the scope surrounding let-syntax, while with letrec-syntax the bindings of names are included in the scope where transformer-spec is evaluated. Thus letrec-syntax allows mutually recursive macros.

Transformer specs
The transformer-spec is a special expression that evaluates to a macro transformer. It is evaluated in a different phase than the other expressions, since macro transformers must be executed during compiling. So there are some restrictions.

At this moment, only one of the following expressions are allowed:

1. A syntax-rules form. This is called “high-level” macro, for it uses pattern matching entirely, which is basically a different declarative language from Scheme, thus putting the complication of the phasing and hygiene issues completely under the hood. Some kind of macros are easier to write in syntax-rules. See Section 5.2.1 [Syntax-rules macro transformer], page 91, for further description.

2. An er-macro-transformer form. This employs explicit-renaming (ER) macro, where you can use arbitrary Scheme code to transform the program, with required renaming to keep hygienity. The legacy Lisp macro can also be written with ER macro if you don’t use renaming. See Section 5.2.2 [Explicit-renaming macro transformer], page 93, for the details.

3. Macro or syntax keyword. This is Gauche’s extension, and can be used to define alias of existing macro or syntax keyword.

(define-syntax si if)
(define écrivez write)

(si (< 2 3) (écrittez "oui"))

### 5.2.1 Syntax-rules macro transformer

**syntax-rules** (literal ...) clause clause2 ...

**syntax-rules** ellipsis (literal ...) clause clause2 ...

[R7RS base] This form creates a macro transformer by pattern matching. Each clause has the following form:

\[(\text{pattern template})\]

A pattern denotes a pattern to be matched to the macro call. It is an S-expression that matches if the macro call has the same structure, except that symbols in pattern can match a whole subtree of the input; the matched symbol is called a pattern variable, and can be referenced in the template.

For example, if a pattern is \((_ "foo" (a b))\), it can match the macro call \((x "foo" (1 2))\), or \((x "foo" (1 (2 3)))\), but does not match \((x "bar" (1 2))\), \((x "foo" (1))\) or \((x "foo" (1 2) 3)\). You can also match repeating structure or literal symbols; we’ll discuss it fully later.

Clauses are examined in order to see if the macro call form matches its pattern. If matching pattern is found, the corresponding template replaces the macro call form. A pattern variable in the template is replaced with the subtree of input that is bound to the pattern variable.

Here’s a definition of `when` macro in Section 5.1 [Why hygienic?], page 87, using syntax-rules:

\[
\text{(define-syntax when}
\text{(syntax-rules ()}
\text{[(_ test body ...) (if test (begin body ...))])})
\]

The pattern is \((_ test body ...)\), and the template is \((if test (begin body ...))\). The ellipsis \(\ldots\) is a symbol; we’re not omitting code here. It denotes that the previous pattern \(body\) may repeat zero or more times.

So, if the `when` macro is called as \((\text{when (zero? x) (print "huh?") (print "we got zero!")})\), the macro expander first check if the input matches the pattern.

- The test in pattern matches the input \((zero? x)\).
- The body in pattern matches the input \((\text{print "huh?"})\) and \((\text{print "we got zero!"})\).

The matching of body is a bit tricky; as a pattern variable, you may think that body works like an array variable, each element holds each match—and you can use them in similarly repeating substructures in template. Let’s see the template, now that the input fully matched the pattern.

- In the template, if and begin are not pattern variable, since they are not appeared in the pattern. So they are inserted as identifiers—that is, hygienic symbols effectively renamed to make sure to refer to the global if and begin, and will be unaffected by the macro use environment.
- The test in the template is a pattern variable, so it is replaced for the matched value, \((\text{zero? x})\).
- The body is also a pattern variable. The important point is that it is also followed by ellipsis. So we repeat body as many times as the number of matched values. The first value, \((\text{print "huh?"})\), and the second value, \((\text{print "we got zero!"})\), are expanded here.
Hence, we get \((\text{if} \ (\text{zero} \ ? \ x) \ (\text{begin} \ (\text{print} \ "\text{huh}\)?") \ (\text{print} \ "\text{we got zero}\)!"))\) as the result of expansion. (With the note that \text{if} \ and \ \text{begin} \ refers \ to \ the \ identifiers \ visible \ from \ the \ macro \ definition \ environment.\)

The expansion of ellipses is quite powerful. In the template, the ellipses don’t need to follow the sequence-valued pattern variable immediately; the variable can be in a substructure, as long as the substructure itself is followed by an ellipsis. See the following example:

\[
\begin{align*}
&(\text{define-syntax show} \\
&\quad (\text{syntax-rules} () \\
&\quad \quad [(_ \ \text{expr} \ \ldots) \\
&\quad \quad \quad (\text{begin} \\
&\quad \quad \quad \quad (\text{begin} \ (\text{write} \ '\text{expr}) \ (\text{display} \ "\text{=}") \ (\text{write} \ \text{expr}) \ (\text{newline}) \\
&\quad \quad \quad \quad \ldots)])])
\end{align*}
\]

If you call this macro as follows:

\[
(\text{show} \ (+ \ 1 \ 2) \ (/ \ 3 \ 4))
\]

It is expanded to the following form, modulo hygienity:

\[
\begin{align*}
&(\text{begin} \\
&\quad (\text{begin} \ (\text{write} \ '(+ \ 1 \ 2)) \ (\text{display} \ "\text{=}") \ (\text{write} \ (+ \ 1 \ 2)) \ (\text{newline}) \\
&\quad (\text{begin} \ (\text{write} \ '/ \ 3 \ 4)) \ (\text{display} \ "\text{=}") \ (\text{write} \ (/ \ 3 \ 4)) \ (\text{newline}))
\end{align*}
\]

So you’ll get this output.

\[
\begin{align*}
(+ \ 1 \ 2)=3 \\
(/ \ 3 \ 4)=3/4
\end{align*}
\]

You can also match with a repetition of substructures in the pattern. The following example is a simplified \text{let} that expands to \text{lambda}:

\[
\begin{align*}
&(\text{define-syntax my-let} \\
&\quad (\text{syntax-rules} () \\
&\quad \quad [(_ \ ((\text{var} \ \text{init}) \ \ldots) \ \text{body} \ \ldots) \\
&\quad \quad \quad (\text{lambda} \ (\text{var} \ \ldots) \ \text{body} \ \ldots) \ \text{init} \ \ldots)])
\end{align*}
\]

If you call it as \((\text{my-let} \ ((a \ \text{expr1}) \ (b \ \text{expr2})) \ \text{foo})\), then \text{var} \ is \ matched \ to \ \text{a} \ and \ \text{b}, \ while \ \text{init} \ is \ matched \ to \ \text{expr1} \ and \ \text{expr2}, \ respectively. \ They \ can \ be \ used \ separately \ in \ the \ template.

Suppose “level” of a pattern variable means the number of nested ellipses that designate repetition of the pattern variable. A subtemplate can be followed as many ellipses as the maximum level of pattern variables in the subtemplate. In the following example, the level of pattern variable \text{a} \ is \ 1 \ (it \ is \ repeated \ by \ the \ last \ ellipsis \ in \ the \ pattern), \ while \ the \ level \ of \ \text{b} \ is \ 2 \ (repeated \ by \ the \ last \ two \ ellipses), \ and \ the \ level \ of \ \text{c} \ is \ 3 \ (repeated \ by \ all \ the \ ellipses).

\[
\begin{align*}
&(\text{define-syntax ellipsis-test} \\
&\quad (\text{syntax-rules} () \\
&\quad \quad [(_ \ (a \ (b \ c \ \ldots) \ \ldots) \ \ldots) \\
&\quad \quad \quad \cdot(a \ \ldots) \\
&\quad \quad \quad (((a \ b) \ \ldots) \ \ldots) \\
&\quad \quad \quad (((a \ b \ c) \ \ldots) \ \ldots))])
\end{align*}
\]

In this case, the subtemplate \text{a} \ must \ be \ repeated \ by \ one \ level \ of \ ellipsis, \ \text{(a b)} \ must \ be \ repeated \ by \ two, \ and \ \text{(a b c)} \ must \ be \ repeated \ by \ three.

\[
\begin{align*}
&(\text{ellipsis-test} \ (1 \ (2 \ 3 \ 4) \ (5 \ 6)) \ (7 \ (8 \ 9 \ 10 \ 11))) \\
&\Rightarrow ((1 \ 7) \\
&\quad (((1 \ 2) \ (1 \ 5)) \ ((7 \ 8))) \\
&\quad (((1 \ 2 \ 3) \ (1 \ 2 \ 4)) \ ((1 \ 5 \ 6))) \ ((7 \ 8 \ 9) \ (7 \ 8 \ 10) \ (7 \ 8 \ 11)))
\end{align*}
\]
In the template, more than one ellipsis directly follow a subtemplate, splicing the leaves into the surrounding list:

```
(define-syntax my-append
  (syntax-rules ()
    [(_ (a ...) ...) '(a ... ...)])
)
```

```
(my-append (1 2 3) (4) (5 6))
⇒ (1 2 3 4 5 6)
```

```
(define-syntax my-append2
  (syntax-rules ()
    [(_ ((a ...) ...) ...) '(a ... ... ...)])
)
```

```
(my-append2 ((1 2) (3 4)) ((5) (6 7 8)))
⇒ (1 2 3 4 5 6 7 8)
```

Note: Allowing multiple ellipses to directly follow a subtemplate, and a pattern variable in a subtemplate to be enclosed within more than the variable’s level of nesting of ellipses, are extension to R7RS, and defined in SRFI-149. In the above examples, ellipsis-test, my-append and my-append2 are outside of R7RS.

Identifiers in a pattern is treated as pattern variables. But sometimes you want to match a specific identifier in the input. For example, the built-in cond and case detects an identifier else as a special identifier. You can use literal . . . for that. See the following example.

```
(define-syntax if+
  (syntax-rules (then else)
    [(_ test then expr1 else expr2) (if test expr1 expr2)])
)
```

The identifiers listed as the literals don’t become pattern variables, but literally match the input. If the input doesn’t have the same identifier in the position, match fails.

```
(if+ (even? x) then (/ x 2) else (/ (+ x 1) 2))
```

expands into

```
(if (even? x) (/ x 2) (/ (+ x 1) 2))
```

```
(if+ (even? x) foo (/ x 2) bar (/ (+ x 1) 2))
⇒ ERROR: malformed if+
```

We’ve been saying identifiers instead of symbols. Roughly speaking, an identifier is a symbol with the surrounding syntactic environment, so that they can keep identity under renaming of hygiene macro.

The following example fails, because the else passed to the if+ macro is the one locally bound by let, which is different from the global else when if+ was defined, hence they don’t match.

```
(let ((else #f))
  (if+ (even? x) then (/ x 2) else (/ (+ x 1) 2))
⇒ ERROR: malformed if+
```

5.2.2 Explicit-renaming macro transformer

Er-macro-transformer procedure-expr

[Special Form]

Creates a macro transformer from the given procedure-expr. The created macro transformer has to be bound to the syntactic keyword by define-syntax, let-syntax or letrec-syntax. Other use of macro transformers is undefined.
Chapter 5: Macros

The *procedure-expr* must evaluate to a procedure that takes three arguments; *form*, *rename* and *id=?*. The *form* argument receives the S-expression of the macro call. The *procedure-expr* must return an S-expression as the result of macro expansion. This part is pretty much like the traditional lisp macro. In fact, if you ignore *rename* and *id=?*, the semantics is the same as the traditional (unhygienic) macro. See the following example (Note the use of *match*; it is a good tool to decompose macro input):

```
(use util.match)

;; Unhygienic 'when-not' macro
(define-syntax when-not
  (er-macro-transformer
   (^[form rename id=?]
    (match form
     [(_ test expr1 expr ...)
      '(if (not ,test) (begin ,expr1 ,@expr))
     [_ (error "malformed when-not:" form)]))))

(macroexpand '(when-not (foo) (print "a") 'boo))
⇒ (if (not (foo)) (begin (print "a") 'boo))
```

This is ok as long as you know you don’t need hygiene—e.g. when you only use this macro locally in your code, knowing all the macro call site won’t contain name conflicts. However, if you provide your *when-not* macro for general use, you have to protect namespace pollution around the macro use. For example, you want to make sure your macro work even if it is used as follows:

```
(let ((not values))
  (when-not #t (print "This shouldn’t be printed")))
```

The *rename* argument passed to *procedure-expr* is a procedure that takes a symbol (or, to be precise, a symbol or an identifier) and effectively renames it to a unique identifier that keeps identity within the macro definition environment and won’t be affected in the macro use environment.

As a rule of thumb, you have to pass *all new identifiers you insert into macro output* to the *rename* procedure to keep hygiene. In our *when-not* macro, we insert *if*, *not* and *begin* into the macro output, so our hygienic macro would look like this:

```
(define-syntax when-not
  (er-macro-transformer
   (^[form rename id=?]
    (match form
     [(_ test expr1 expr ...)
      '((rename 'if) ,,(rename 'not) ,test)
     [(,(rename 'begin) ,expr1 ,@expr))
     [_ (error "malformed when-not:" form)]))))
```

This is cumbersome and makes it hard to read the macro, so Gauche provides an auxiliary macro *quasirename*, which works like *quasiquote* but renaming identifiers in the form. See the entry of *quasirename* below for the details. You can write the hygienic *when-not* as follows:

```
(define-syntax when-not
  (er-macro-transformer
   (^[form rename id=?]
    (match form
```
You can intentionally break hygiene by inserting a symbol without renaming. The following code implements anaphoric when, meaning the result of the test expression is available in the expr1 exprs... with the name it. Since the binding of the identifier it does not exist in the macro use site, but rather injected into the macro use site by the macro expander, it is unhygienic.

```
(define-syntax awhen
  (er-macro-transformer
   (^[form rename id=?]
    (match form
     [[_ test expr1 expr ...]
      (quasirename rename
       '(if (not ,test) (begin ,expr1 ,@expr))
       [._ (error "malformed when-not:" form)])))))
```

If you use quasirename, you can write ,it to prevent it from being renamed:

```
(define-syntax awhen
  (er-macro-transformer
   (^[form rename id=?]
    (match form
     [[_ test expr1 expr ...]
      (quasirename rename
       '(let1 ,'it ,test
         (begin ,expr1 ,@expr))))))))
```

Here's an example:

```
(awhen (find odd? '(0 2 8 7 4))
  (print "Found odd number:" it))
⇒ prints Found odd number:7
```

Finally, the id=? argument to the procedure-expr is a procedure that takes two arguments, and returns #t iff both are identifiers and either both are referring to the same binding or both are free. It can be used to compare literal syntactic keyword (e.g. else in cond and case forms) hygienically.

The following if=> macro behaves like if, except that it accepts (if=> test => procedure) syntax, in which procedure is called with the value of test if it is not false (similar to (cond [test => procedure]) syntax). The symbol => must match hygienically, that is, it must refer to the same binding as in the macro definition.

```
(define-syntax if=>
  (er-macro-transformer
   (^[form rename id=?]
    (match form
     [[_ test a b]
      (if (id=? (rename '=>) a)
       (quasirename rename
        '(let ((t ,test))
          (if t ,(b t)))
        (quasirename rename
         '(if ,test ,a ,b))))])))
```

The call `(rename '=>)` returns an identifier that captures the binding of `=>` in the macro definition, and using `id=?` with the thing passed to the macro argument checks if both refer to the same binding.

```scheme
(if=> 3 => list) ⇒ (3)
(if=> #f => list) ⇒ #<undef>
```

`; If the second argument isn’t =>, if=> behaves like ordinary if:

```scheme
(if=> #t 1 2) ⇒ 1
```

`; The binding of => in macro use environment differs from

`; the macro definition environment, so this if=> behaves like

`; ordinary if, instead of recognizing literal =>.

```scheme
(let ((=> 'oof)) (if=> 3 => list)) ⇒ oof
```

---

### quasirename renamer quasiquoted-form

[Macro]

It works like quasiquote, except that the symbols and identifiers that appear in the “literal” portion of `form` (i.e. outside of `unquote` and `unquote-splicing`) are replaced by the result of applying `rename` on themselves.

The `quasiquoted-form` argument must be a quasiquoted form. The outermost quasiquote ‘ is consumed by `quasirename` and won’t appear in the output. The reason we require it is to make nested quasiquotes/quasirenames work.

For example, a form:

```scheme
(quasirename r ‘(a ,b c “d”))
```

would be equivalent to write:

```scheme
(list (r ’a) b (r ’c) “d”)
```

This is not specifically tied to macros; the `renamer` can be any procedure that takes one symbol or identifier argument:

```scheme
(quasirename (^[x] (symbol-append ’x: x)) ‘(+ a ,(+ 1 2) 5)) ⇒ (x:+ x:a 3 5)
```

However, it comes pretty handy to construct the result form in ER macros. Compare the following two:

```scheme
(use util.match)

;; using quasirename
(define-syntact syntax swap
  (er-macro-transformer
   (^[f r c]
    (match f
     [(_ a b) (quasirename r
      ‘(let ((tmp ,a))
       (set! ,a ,b)
       (set! ,b tmp)))]))))

;; not using quasirename
(define-syntact syntax swap
  (er-macro-transformer
   (^[f r c]
    (match f
     [(_ a b) ‘((r’let) (((r’tmp) ,a))
       ((r’set!) ,a ,b)]))])})
```
Note: In Gauche 0.9.7 and before, `quasirename` didn’t use quasiquoted form as the second argument; you can write `(quasirename r form)` instead of `(quasirename r 'form)`.

For the backward compatibility, we support the form without quasiquote by default for a while.

If you already have a quasirename form that does intend to produce a quasiquoted form, you have to rewrite it with double quasiquote: `(quasirename r "form")`.

To help transition, the handling of quasiquote in of `quasirename` can be customized with the environment variable `GAUCHE_QUASIRENAME_MODE`. It can have one of the following values:

- **legacy** `Quasirename` behaves the same way as 0.9.7 and before; use this to run code for 0.9.7 without any change.
- **compatible** `Quasirename` behaves as described in this entry; if `form` lacks a quasiquote, it silently assumes one. Existing code should work, except the rare case when you intend to return a quasiquoted form.
- **warn** `Quasirename` behaves as described in this entry, but warns if `form` lacks a quasiquote.
- **strict** `Quasirename` raises an error if `form` lacks a quasiquote. This will be the default behavior in future.

### 5.3 Traditional macros

**define-macro** `name procedure` [Special Form]

**define-macro** `(name . formals) body . . .` [Special Form]

Defines `name` to be a global macro whose transformer is `procedure`. The second form is a shorthand notation of the following form:

```
(define-macro name (lambda formals body ...))
```

When a form `(name arg . . .)` is seen by the compiler, it calls `procedure` with `arg . . .`. When `procedure` returns, the compiler inserts the returned form in place of the original form, and compile it again.

To avoid name conflict with the bindings inserted by the macro, you can use `gensym`, just like traditional Lisp macros (see Section 6.7 [Symbols], page 155).

```
(define-macro (if-let1 var test then else)
  (let1 tmp (gensym)
    '(let ((,tmp ,test))
      (if ,tmp ,then ,else))))
```

```
(macroexpand ' (if-let1 v (find odd? '(2 4 6 7 8))
  (* v v)
  #f))
⇒ (let ((#:G1013 (find odd? (quote (2 4 6 7 8))))
  (if #:G1013 (* v v) #f))
```

Note that `gensym` can’t protect name conflict with global bindings inserted by the macro. Section 5.1 [Why hygienic?], page 87, discusses this issue.
5.4 Hybrid macros

A hybrid macro is both a macro and a procedure simultaneously. If a symbol bound to a hybrid macro appears in the first position of a form, it behaves like a macro and the form is expanded according to its macro expander. If a symbol appears other places, it is evaluated to a procedure at runtime.

It can realize so-called “compiler macros”—at a compile time, the macro part examines the arguments and can transform the form as desired. In all other circumstances, it behaves like a normal procedure binding, so you can pass the procedure to map, for example.

\texttt{define-hybrid-syntax variable expr transformer-spec}\quad [Macro]

Binds \texttt{variable} to both an ordinary Scheme value and a macro simultaneously. At the compile time, \texttt{transformer-spec} is evaluated; it must yield a macro in the the compile-time environment, and bound to \texttt{variable} to be used at macro expansion. At the execution time, \texttt{expr} is evaluated and bound to \texttt{variable} to be used as a run-time value.

The macro transformer can return the input form as is (that is, returns an object \texttt{eq?} to the input form), to indicate that it doesn’t need to expand it. In that case, Gauche compiles the form as an ordinary procedure call, to use the value of \texttt{expr} at run-time.

Note: If what you want to do with the hybrid macro is just to inline-expand the procedure body, use \texttt{define-inline} (see Section 4.10 [Definitions], page 68).

Note about the syntax: Traditionally in Lisp, compiler macros are defined by a separate form from the procedure binding.

However, having bindings to the same identifier twice makes the program semantics ambiguous. What if the two forms are separated into different modules? What if the identifier is redefined? It would be clearer that single form determines the binding.

5.5 Identifiers

In the discussion of hygienic macros, we keep saying the symbols are \textit{effectively renamed}. What it means is that we don’t actually create a new symbol with a new name. We have to remember the origin of the renamed symbol to resolve the scope of the variable, and having a separate table to keep track of renamed symbols would be costly. Instead, the “rename” procedure wraps the symbols in the input with syntactic information.

When you play with macro internals, you’ll see an object that is printed something like \#<identifier user#foo.fb4ca828>. That’s the wrapped symbol.

If one macro output is passed to another macro expander, the wrapped symbol may further be wrapped.

The macro expander must assume that symbols in the input are already wrapped by another macro expander. So, instead of calling it a “symbol”, we call it an “identifier”. An identifier is something that usually works as a variable or a syntactic keyword in the program. It may be a symbol or a wrapped identifier. (Note that symbols in quoted literals are bare symbols, for the \texttt{quote} form strips wrappers.)

Legacy Lisp macros sometimes examines the symbols in the input form. In Scheme, you have to treat the input program as a tree of identifiers and other objects. You can test whether an object is an identifier or not by \texttt{identifier?}, where traditional Lisp macro would have used \texttt{symbol?}. To compare identifiers, you need to use the “compare” procedure passed to the \texttt{er-macro expander}, or \texttt{free-identifier=?}.

\texttt{<identifier>}\quad [Builtin Class]

A class of wrapped identifier. It is created as a result of “renaming” in the hygienic macro expander.
A wrapped identifier contains transient information about the program source, and cannot be portably saved or passed around; it is only valid in the macro expansion phase. For the details of identifier, see Section 5.5 [Identifiers], page 98.

**identifier? obj** [Function]
Returns #t if *obj* is either a symbol or a wrapped identifier. Returns #f otherwise.

Note: In R6RS, `identifier?` only returns #t for an identifier object, which is of a disjoint type from symbols. You can use `wrapped-identifier?` below to check if an object is an identifier other than a symbol.

**wrapped-identifier? obj** [Function]
Returns #t iff *obj* is a wrapped identifier.

This is R6RS's `identifier?`.

**identifier->symbol obj** [Function]
Returns the symbol that is the origin of the *obj*, which must be either a symbol or a wrapped identifier.

**free-identifier=? id1 id2** [Function]
When both arguments *id1* and *id2* are wrapped identifiers, returns #t if either (1) *id1* and *id2* both refer to the same binding, or (2) *id1* and *id2* are both unbound. Otherwise, #f is returned.

If at least one of *id1* or *id2* is not a wrapped identifier, #f is returned. Note that bare symbols can’t be compared with this procedure, for they lack the necessary lexical information. To obtain a wrapped identifier, you need to pass a bare symbol to the “rename” procedure passed to the er-macro transformer.

Usually you don’t need to use this procedure directly, for the compare procedure passed to the er-macro transformer is suffice.

**unwrap-syntax form** [Function]
Returns a copy of *form*, except removing wrappings of identifiers in it. The output of macro expanders contain wrapped identifiers, which is bothersome to see. This procedure traverses *form* and replaces any wrapped identifiers with its original symbol, retrieved by `identifier->symbol`.

Note that, although the result is an ordinary S-expression easier to read, syntactic information is completely lost. For example, distinct identifiers can become indistinguishable if they happen to have the same name (it happens often when you generate temporary variables via recursive calls of `syntax-rules`). If you distinguish newly inserted identifiers with the same name, use `unravel-syntax`.

**unravel-syntax form** [Function]
Returns a copy of *form* while removing wrappings of identifiers in it, but attach suffix if two distinct identifiers have the same name, so that they won’t be confused.

For example, a common idiom of `syntax-rules` to generate temporary variables with recursions create all variables with the same name, although each variable is different because they are inserted by the different invocation of the expander. If you pass its output to `unwrap-syntax`, all syntactic information is stripped and these variables can’t be distinguished from one another.

This procedure is automatically called with some macro utilities; See Section 5.6.1 [Tracing macro expansion], page 100, and see Section 5.6.2 [Expanding macros manually], page 102, for the example of output of `unravel-syntax`.

Note that the global identifiers becomes bare symbols, so you are still unable to tell which module the global identifiers refer to.
5.6 Debugging macros

Macro expansion happens at the compile time, which makes it difficult to debug. The best way to avoid headache of macro debugging is not to write macros unless they’re absolutely necessary, and keep them as simple as possible if you need to write ones.

However, if you find yourself in an unfortunate situation that you have to untangle hairy macros, Gauche has some tools to help.

5.6.1 Tracing macro expansion

Macro tracing shows the input to the macro expander and the result of its expansion on selected macros. Suppose you have the following macro definition. It’s essentially the same as shown in the definition of letrec in R7RS section 7.3:

```
(define-syntax my-letrec
  (syntax-rules ()
    [(_ ((var init) ...) body ...)
      (my-letrec "tmps" (var ...) () ((var init) ...) body ...)]
    [(_ "tmps" () (tmp ...) ((var init) ...) body ...)
      (let ((var 'undefined) ...)
        (let ((tmp init) ...)
          (set! var tmp) ...
          body ...))]
    [(_ "tmps" (x y ...) (tmp ...) binds body ...)
      (my-letrec "tmps" (y ...) (newtmp tmp ...) binds body ...)])
```

The my-letrec macro uses an idiom to generate temporary variables by looping with "tmps" tag. You can see how the macro is expanded step by step, by tracing my-letrec:

```
gosh> (trace-macro 'my-letrec)
(my-letrec)
gosh> (my-letrec [(ev? (^n (if (= n 0) #t (od? (- n 1))))
            (od? (^n (if (= n 0) #f (ev? (- n 1))))))
            (ev? 3))
Macro input>>>
(my-letrec
  ((ev? (^n (if (= n 0) #t (od? (- n 1))))
    (od? (^n (if (= n 0) #f (ev? (- n 1))))))
    (ev? 3))
Macro output<<<
(my-letrec
  "tmps"
  (ev? od?)
()  
  ((ev? (^n (if (= n 0) #t (od? (- n 1))))
    (od? (^n (if (= n 0) #f (ev? (- n 1))))))
    (ev? 3))
Macro input>>>
(my-letrec
  "tmps"
  (ev? od?)
()  
  ((ev? (^n (if (= n 0) #t (od? (- n 1))))
    (od? (^n (if (= n 0) #f (ev? (- n 1))))))
    (ev? 3))
```

(od? (~n (if (= n 0) #f (ev? (~ n 1)))))
(ev? 3))

Macro output<<<
(my-letrec
 "tmps"
 (od?)
 (newtmp.0)
 ((ev? (~n (if (= n 0) #t (od? (~ n 1)))))
  (od? (~n (if (= n 0) #f (ev? (~ n 1)))))
  (ev? 3))

Macro input>>>
(my-letrec
 "tmps"
 (od?)
 (newtmp.0)
 ((ev? (~n (if (= n 0) #t (od? (~ n 1)))))
  (od? (~n (if (= n 0) #f (ev? (~ n 1)))))
  (ev? 3))

Macro output<<<
(my-letrec
 "tmps"
 ()
 (newtmp.1 newtmp.0)
 ((ev? (~n (if (= n 0) #t (od? (~ n 1)))))
  (od? (~n (if (= n 0) #f (ev? (~ n 1)))))
  (ev? 3))

Macro input>>>
(my-letrec
 "tmps"
 ()
 (newtmp.0 newtmp.1)
 ((ev? (~n (if (= n 0) #t (od? (~ n 1)))))
  (od? (~n (if (= n 0) #f (ev? (~ n 1)))))
  (ev? 3))

Macro output<<<
(let
 ((ev? (quote undefined)) (od? (quote undefined)))
 (let
  ((newtmp.0 (~n (if (= n 0) #t (od? (~ n 1)))))
   (newtmp.1 (~n (if (= n 0) #f (ev? (~ n 1)))))
   (set! ev? newtmp.0)
   (set! od? newtmp.1)
   (ev? 3)))

#f
In the above example, the S-expressions after `gosh>` prompt is what you type; all other things are Gauche’s answer, including Macro input and Macro output S-expressions.

The S-expression shown with Macro input is the input of the macro expander, and the one with Macro output is the expanded result. Actual macro output has syntactic information attached, but the tracer strips them off for the legibility.

Note that the loop introduces new temporary variables with the same name (newtemp), but they are treated as different identifiers in the macro expansion.

Once you’re done debugging, don’t forget to call `untrace-macro` with no arguments to remove macro traces. If there’s a macro trace set, all macro expansions get some overhead, so don’t leave macro traces.

```scheme
(gosh> (untrace-macro))
#f
```

**trace-macro**

[Function]

**trace-macro boolean**

Get/set current macro trace setting. Macro trace setting can be one of the following values:

- **#f**  
  Macro tracing is off. This is the default setting.

- **#t**  
  All macro expansions are traced.

- **(name-or-pattern ...)**  
  Trace macros that match any one of name-or-pattern, which is either a symbol or a regexp. If it’s a symbol, a macro whose name is the same as the symbol is traced. If it’s a regexp, macros whose name match the regexp are traced.

When called without arguments, `trace-macro` doesn’t change the setting; it returns the current setting.

When called with single boolean value, it sets the current setting to that value. Returns the updated setting.

When called with one or more name-or-pattern, it adds them to the current setting. Note that if the current setting is `#t`, it remains `#t`, for all macros are already traced. Returns the updated setting.

If macro trace settings is not `#f`, it incurs overhead for every macro expansion. Be careful not to leave macro trace set.

The trace information is output to the current trace port. (see Section 6.21.3 [Common port operations], page 260).

**untrace-macro**

[Function]

**untrace-macro name-or-pattern ...**

When called without arguments, it turns macro trace off.

When called with one or more name-or-pattern, which is either a symbol or a regexp, `untrace-macro` removes them from the currently traced macros. Note that if the current macro trace setting is `#t` (trace all macros), you can’t remove traced macro individually.

It returns the updated macro trace setting.

### 5.6.2 Expanding macros manually

**macroexpand form :optional env**

[Function]

**macroexpand-1 form :optional env**

[Function]

If `form` is a list and its first element is a variable globally bound to a macro, `macroexpand-1` invokes its macro transformer and returns the expanded form. Otherwise, returns `form` as is.
macroexpand repeats macroexpand-1 until the outermost expression of form can’t be expanded. (It doesn’t expand macros other than outermost one. If you want to expand all the macros within form, use macroexpand-all).

These procedures can be used to expand globally defined macros.

Internally, hygienic macro expansion wraps symbols in form with syntactic information to keep hygiene. However, such information is hard to read, and not suitable when you just want to expand a macro in REPL to check its result. So, by default, these procedures strips syntactic information. For the identifiers introduced in the macro, it renames them to avoid name conflicts.

The following example expands my-letrec macro (see Section 5.6.1 [Tracing macro expansion], page 100, for the definition) and results shows temporary variable introduced by the macro (newtemp) to be renamed.

```
(macroexpand
  '(my-letrec [(ev? (^n (if (= n 0) #t (od? (- n 1)))))
               (od? (^n (if (= n 0) #f (ev? (- n 1))))))
             (ev? 3)))
⇒
(let
  ((ev? (quote undefined)) (od? (quote undefined)))
  (let
    ((newtmp.0 (^n (if (= n 0) #t (od? (- n 1)))))
     (newtmp.1 (^n (if (= n 0) #f (ev? (- n 1)))))
     (set! ev? newtmp.0)
     (set! od? newtmp.1)
     (ev? 3)))
```

If you pass a module to the env argument, it is used as the macro use environment. You can also pass #t to let it use the current runtime environment as the macro use environment. In those cases, syntactic information in the output won’t be stripped.

If you want to use the output of macroexpand as a program, e.g. embed it into another macro expansion, you need syntactic information preserved.

macroexpand-all form :optional env

Fully expand macros inside form. The result only contains function calls and Gauche’s built-in syntax.

By default, or #t is passed to env, the form is assumed to be a toplevel form within the current runtime module. You can also pass a module to env to specify the alternative toplevel environment.

Any local variables introduced in form is renamed to avoid collision. Since each local variable has unique name, all let forms become letrec forms (we can safely replace let with letrec if no bindings introduced by let shadows outer bindings.)

NB: If a macro in form inserts a reference to a global variable which belongs to other module, the information is lost in the current implementation. There are a few ways to address this issue; we may leave such reference as an identifier object, convert it to with-module form, or introduce a special syntax to represent such case. It’s undecided currently, so do not rely too much on the current behavior. For the time being, it’s best to use this feature only for interactive macro testing.

```
(macroexpand-all
  '(letrec-syntax
```
[(when-not (syntax-rules ()
  [(_ test . body) (if test #f (begin . body))])]]
(let ([if list])
  (define x (expt foo))
  (let1 x 3
    (when-not (bar) (if x)))))
⇒ (letrec ((if.0 list))
  (letrec ((x.1 (expt foo)))
    (letrec ((x.2 '3))
      (if (bar) '#f (if.0 x.2)))))

%macroexpand form
%macroexpand-1 form

5.7 Macro utilities

syntax-error msg arg ...
syntax-errorf fmt arg ...

Signal an error. They are same as error and errorf (see Section 6.19.2 [Signaling exceptions], page 248), except that the error is signaled at macro-expansion time (i.e. compile time) rather than run time.

They are useful to tell the user the wrong usage of macro in the comprehensive way, instead of the cryptic error from the macro transformer. Because of the purpose, arg ... are first passed to unwrap-syntax to strip off the internal syntactic binding informations (see Section 5.5 [Identifiers], page 98).

(define-syntax my-macro
  (syntax-rules ()
    ((_ a b) (foo2 a b))
    ((_ a b c) (foo3 a b c))
    ((_ . ?)
      (syntax-error "malformed my-macro" (my-macro . ?))))))

(my-macro 1 2 3 4)
⇒ error: "malformed my-macro: (my-macro 1 2 3 4)"
6 Core library

6.1 Types and classes

Scheme is a dynamically and strongly typed language. That is, every value knows its type at run-time, and the type determines what kind of operations can be applied on the value.

The Scheme standard is pretty simple on types; basically, an object being a type means the type predicate returns true on the object, and that’s all. Gauche adopts a bit more elaborated system—types are first-class objects and you can query various information.

In Gauche, types are conventionally named with brackets `<` and `>`, e.g. `<string>`. It’s nothing syntactically special with these brackets; they’re valid characters to consist of variable names.

### 6.1.1 Prescriptive and descriptive types

Types are used in two ways. A type can be seen as a template of the actual values (instances)—that is, a type prescribes the structure of, and possible operations on, its instances. In Gauche, such prescriptive types are represented by classes. Every value in Gauche belongs to a class, which can be queried with the `class-of` procedure. You can also define your own class with `define-class` or `define-record-type` (see Chapter 7 [Object system], page 329, and see Section 9.28 [Record types], page 502).

A type can also be seen as a constraint of a given expression—that is, a type describes what characteristics a certain expression must have. Gauche has entities to represent such descriptive types separate from classes. Descriptive types are created with type constructors (see Section 6.1.4 [Type expressions and type constructors], page 107). We also have a set of predefined descriptive types to communicate to C libraries (see Section 6.1.5 [Native types], page 109).

A value is an instance of a class. For example, 1 is an instance of `<integer>`, and "xyz" is an instance of `<string>`. This prescriptive type relationship is checked with `is-a?` procedure: `(is-a? 1 <integer>)` and `(is-a? "xyz" <string>)` both return `#t`.

On the other hand, you may have a procedure that takes either a number or a string as an argument. “A number or a string” is a type constraint, and can be expressed as a descriptive type, `(</> <number> <string>)`. Descriptive type relationship is checked with `of-type?` procedure: `(of-type? 1 (</> <number> <string>))` and `(of-type? "xyz" (</> <number> <string>))` both returns `#t`. More conveniently, `(assume-type arg (</> <number> <string>))` would raise an error if `arg` doesn’t satisfy the given type constraint.

### 6.1.2 Generic type predicates

A “type predicate” is a predicate that tells if an object is of a specific type; e.g. `number?` tells you if the argument is a number. Since types are first-class in Gauche, we have predicates that can tell an object is of a given type, as well as predicates to ask the relationship between types.

#### is-a? obj class

This is a prescriptive types predicate. Returns true iff `obj` is an instance of `class` or an instance of descendants of `class`.

- `(is-a? 3 <integer>)` ⇒ `#t`
- `(is-a? 3 <real>)` ⇒ `#t`
- `(is-a? 5+3i <real>)` ⇒ `#f`

Note: If `obj`’s class has been redefined, `is-a?` also triggers instance update. See Section 7.2.5 [Class redefinition], page 342, for the details.
of-type? obj type  
This is a descriptive type predicate. Returns true iff obj satisfies the constraints described by type, which can be either a class (in that case, this is the same as is-a?), or a descriptive type (see Section 6.1.4 [Type expressions and type constructors], page 107, below).

(of-type? 1 (</> <number> <string>)) ⇒ #t
(of-type? "a" (</> <number> <string>)) ⇒ #t
(of-type? 'a (</> <number> <string>)) ⇒ #f

subtype? sub super  
Returns #t if a type sub is a subtype of a type super (includes the case that sub is super). Otherwise, returns #f.

In general, if sub is a subtype of super, an object that satisfies the constraints of sub also satisfies the constraints of super, so you can use the object where objects of type super are expected. In other words, sub is more restrictive than super. If both sub and super are classes, sub being a subtype of super means sub is a subclass of super.

(subtype? <integer> <real>) ⇒ #t
(subtype? <char> (</> <char> <string>)) ⇒ #t
(subtype? (</> <integer> <string>) (</> <number> <string>)) ⇒ #t

Note that we’re not rigorous on this “substitution principle”, for we don’t aim at guaranteeing type safety through static analysis. For example, “list of integers” can be used in place of a generic list most of the time, and (subtype? (<List> <integer>) <list>) is #t. However, if list is mutated, you can’t replace generic list with a list of integers—because mutators may try to set non-integer in the list. That kind of cases needs to be handled separately, not on relying solely on subtype?.

subclass? sub super  
Both arguments must be classes. Returns #t iff sub is a subclass of super. A class is regarded as a subclass of itself.

6.1.3 Predefined classes

Predefined classes are bound to a global variable; Gauche’s We’ll introduce classes for each built-in type as we go through this chapter. Here are a few basic classes to start with:

<top>  
This class represents the supertype of all the types in Gauche. That is, for any class X, (subtype? X <top>) is #t, and for any object x, (is-a? x <top>) is #t.

<bottom>  
This class represents the subtype of all the types in Gauche. For any class X, (subtype? <bottom> X) is #t, and for any object x, (is-a? x <bottom>) is #f.

There’s no instance of <bottom>.

Note: Although <bottom> is subtype of other types, the class precedence list (CPL) of <bottom> only contains <bottom> and <top>. It’s because it isn’t always possible to calculate a linear list of all the types. Even if it is possible, it would be expensive to check and update the CPL of <bottom> every time a new class is defined or an existing class is redefined. Procedures subtype? and is-a? treat <bottom> specially.

One of use case of <bottom> is applicable? procedure. See Section 6.15.1 [Procedure class and applicability], page 217.

<object>  
This class represents a supertype of all user-defined classes.
class-of \textit{obj} \quad \text{[Function]}

Returns a class metaobject of \textit{obj}.

\begin{align*}
\text{(class-of 3)} & \Rightarrow \text{#<class <integer>>} \\
\text{(class-of "foo")} & \Rightarrow \text{#<class <string>>} \\
\text{(class-of <integer>)} & \Rightarrow \text{#<class <class>>}
\end{align*}

Note: In Gauche, you can redefine existing user-defined classes. If the new definition has different configuration of the instance, \text{class-of} on existing instance triggers instance updates; see Section 7.2.5 [Class redefinition], page 342, for the details. Using \text{current-class-of} suppresses instance updates (see Section 7.3.2 [Accessing instance], page 346).

### 6.1.4 Type expressions and type constructors

Types are first-class objects manipulatable at runtime in Gauche, but they must be known at compile-time in order to do optimizations or static analysis. In certain places, Gauche requires the the value of type-yielding expressions to be statically computable. We call such expressions \textit{type expressions}.

A type expression is either a global variable reference that are constantly bound to a type, or a call of type constructor:

\[
\begin{align*}
\text{<type-expression>} &: \text{<global-variable-constantly-bound-to-a-type>} \\
\text{\quad | <type-constructor-call>}
\end{align*}
\]

\[
\begin{align*}
\text{<type-constructor-call>} &: (\text{<type-constructor>} \text{<type-constructor-argument>} \ldots)
\end{align*}
\]

\[
\begin{align*}
\text{<type-constructor-argument>} &: \text{<type-expression>} \\
\text{\quad | <integer-constant>} \\
\text{\quad | \text{->}} \\
\text{\quad | *}
\end{align*}
\]

All built-in classes, such as \texttt{<integer>}, are statically bound (in precise terms, they are “inlinable” binding). If you try to alter it, a warning is issued, and the further behavior will be undefined. In future, we’ll make it an error to alter the binding of global variables bound to types. Classes defined with \texttt{define-class} and \texttt{define-record-type} are also bound as inlinable.

Type constructors are special classes whose instances are descriptive types. Type constructors can be invoked as if they are a procedure, but \texttt{<type-constructor-call>} above is recognized by the compiler and the derived type is computed at compile-time. So, at runtime you only see the resulting derived type instance.

For example, \texttt{<?>} is a type constructor that creates “maybe”-like type, e.g. “an integer or \texttt{#f}”. (Do not confuse this with Maybe type defined in SRFI-189, see Section 11.42 [Maybe and Either optional container types], page 765). A type expression \texttt{(<?> <integer>)} yields such type (printed as \texttt{#<? <integer>>}):

\[
\begin{align*}
\text{(<?> <integer>)} & \Rightarrow \text{#<? <integer>>}
\end{align*}
\]

It looks like a procedure call, but it's computed at compile-time:

\begin{verbatim}
gosh> (disasm (^[] (<?> <integer>)))
CLOSURE #<closure (#f)>
=== main_code (name=#f, cc=0x7f5cd76ab540, codevec=...):
signatureInfo: ((#f))
  0 CONST-RET #<? <integer>>
\end{verbatim}

<?> \texttt{type} \quad \text{[Type Constructor]}

Type must be a type expression. This creates a maybe-type, that is, the object is either of \texttt{type} or \texttt{#f}. Usually, \texttt{#f} indicates that the value is invalid (e.g. the return value of \texttt{assoc}).
This type has ambiguity when \#f can be a meaningful value, but traditionally been used a lot in Scheme, for it is lightweight. We also have a “proper” Maybe type in SRFI-189 (see Section 11.42 [Maybe and Either optional container types], page 765) but that involves extra allocation to wrap the value.

So, those who came from functional programming may find this isn’t a right abstraction, but it has its place in the Scheme world.

</> type ...

[Type Constructor]

Type ... must be type expressions. This creates a sum type of type ... For example, (</> <string> <symbol>)) is a type that is either a string or a symbol.

<Tuple> type ...

[Type Constructor]

Type ... must be type expressions, except that the last argument that may be an identifier *. This creates a product type of type ... .

Actually, this type is looser than the product types in typical statically-typed languages. Our tuple is a subtype of list, with types of each positional element are restricted with type ... . For example, (<Tuple> <integer> <string>) is a list of two elements, the first one being an integer and the second being a string.

(of-type? '(3 "abc") (<Tuple> <integer> <string>)) ⇒ #t

If you need a more strict and disjoint product type, you can just create a class or a record.

If the last argument is *, the resulting type allows extra elements after the typed elements.

(of-type? '(3 "abc" 1 2) (<Tuple> <integer> <string> *)) ⇒ #t

<^> type ... -> type ...

[Type Constructor]

Each type must be type expressions, except that the one right before -> and the last one may be an identifier *. This creates a procedure type, with the constraints in arguments and return types. The type ... before -> are the argument types, and the ones after -> are the result types.

The * in the last of argument type list and/or result type list indicates extra elements are allowed.

In vanilla Scheme, all procedures belong to just one type, that responds true to procedure?. It is simple and flexible, but sometimes the resolution is too coarse to do reasoning on the program.

In Gauche, we can attach more detailed type information in procedures. In the current version, some built-in procedures already have such type information, that can be retrieved with procedure-type:

(procedure-type cons) ⇒ #<^ <top> <top> -> <pair>>

Not all procedures have such information, though. Do not expect the rigorousness of statically typed languages.

At this moment, Scheme-defined procedures treats all argument types as <top>. We’ll provide a way to attach type info in future.

<List> type :optional min-length max-length

[Type Constructor]

Type must be a type expression. These create a list or a vector type whose elements are of type, respectively. For example, (<List> <string>) is a list of strings.

The optional arguments must be a literal real numbers that limit the minimum and maximum (inclusive) length of the list or the vector. When omitted, min-length is zero and max-length is inf.0.

Note: subtype? uses covariant subtyping, that is, (subtype? (<List> <string>) <list>) is #t.
6.1.5 Native types

Native types are a set of predefined descriptive types that bridge Scheme types and C types. They can be used to access binary blobs passed from a foreign functions, for example. They’ll also be used for FFI.

An example of native types is `<int16>`, which corresponds to C’s `int16_t`. You can use `<int16>` to check if a Scheme object can be treated as that type:

```
(of-type? 357 <int16>) ⇒ #t
(of-type? 50000 <int16>) ⇒ #f
```

Or obtain size and alignemnt info on the running platform:

```
gosh> (describe <int16>)
#<native-type <int16>> is an instance of class <native-type>
slots:
  name : <int16>
  super : #<class <integer>>
  c-type-name: "int16_t"
  size : 2
  alignment : 2
```

Predefined native types. The correspondence of Scheme and C types are shown below:

<table>
<thead>
<tr>
<th>Native type</th>
<th>Scheme</th>
<th>C</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;fixnum&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>ScmSmallInt</td>
<td>Integers within fixnum range</td>
</tr>
<tr>
<td><code>&lt;int&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>int</td>
<td>Integers representable in C</td>
</tr>
<tr>
<td><code>&lt;int8&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>int8_t</td>
<td></td>
</tr>
<tr>
<td><code>&lt;int16&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>int16_t</td>
<td></td>
</tr>
<tr>
<td><code>&lt;int32&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>int32_t</td>
<td></td>
</tr>
<tr>
<td><code>&lt;int64&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>int64_t</td>
<td></td>
</tr>
<tr>
<td><code>&lt;short&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td>short</td>
<td></td>
</tr>
<tr>
<td><code>&lt;uint&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;uint8&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;uint16&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;uint32&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;uint64&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ushort&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ulong&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;float&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;double&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;size_t&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ssize_t&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ptrdiff_t&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;off_t&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;void&gt;</code></td>
<td><code>&lt;integer&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6: Core library

### 6.2 Equality and comparison

Comparing two objects seems trivial, but if you look into deeper, there are lots of subtleties hidden in the corners. What should it mean if two procedures are equal to each other? How to order two complex numbers? It all depends on your purpose; there’s no single generic answer. So Scheme (and Gauche) provides several options, as well as the way to make your own.

#### 6.2.1 Equality

Scheme has three different general equality test predicates. Other than these, some types have their own comparison predicates.

**eq? obj1 obj2**

[Function] 
[R7RS base] This is the fastest and finest predicate. Returns #t if obj1 and obj2 are identical objects—that is, if they represent the same object on memory or in a register. Notably, you can compare two symbols or two keywords with eq? to check if they are the same or not. You can think eq? as a pointer comparison for any heap-allocated objects.

Booleans can be compared with eq?, but you can’t compare characters and numbers reliably—objects with the same numerical value may or may not eq? to each other. If you identity comparison needs to include those objects, use eqv? below.

\[
\begin{align*}
(eq? \#t \#t) & \Rightarrow \#t \\
(eq? \#t \#f) & \Rightarrow \#f \\
(eq? 'a 'a) & \Rightarrow \#t \\
(eq? 'a 'b) & \Rightarrow \#f \\
(eq? (list 'a) (list 'a)) & \Rightarrow \#f \\
(let ((x (list 'a))) (eq? x x)) & \Rightarrow \#t
\end{align*}
\]

**eqv? obj1 obj2**

[Function] 
[R7RS base] When obj1 and obj2 are both exact or both inexact numbers (except NaN or -0.0), eqv? returns #t iff (= obj1 obj2) is true. -0.0 is only eqv? to -0.0.

When obj1 and obj2 are both characters, eqv? returns #t iff (char=? obj1 obj2) is true. Otherwise, eqv? is the same as eq? on Gauche.

\[
(eqv? \#\a \#\a) \Rightarrow \#t
\]
(eqv? #\a #\b) ⇒ #f
(eqv? 1.0 1.0) ⇒ #t
(eqv? 1 1) ⇒ #t
(eqv? 1 1.0) ⇒ #f
(eqv? (list 'a) (list 'a)) ⇒ #f
(let ((x (list 'a)))
  (eqv? x x)) ⇒ #t

Note that comparison of NaNs has some peculiarity. Any numeric comparison fails if there’s at least one NaN in its argument. Therefore, (= +nan.0 +nan.0) is always #f. However, Gauche may return #t for (eq? +nan.0 +nan.0) or (eqv? +nan.0 +nan.0).

equal? obj1 obj2

[R7RS+] If obj1 and obj2 are both aggregate types, equal? compares its elements recursively. Otherwise, equal? behaves the same as eqv?.

If obj1 and obj2 are not eqv? to each other, not of built-in types, and the class of both objects are the same, equal? calls the generic function object-equal?. By defining the method, users can extend the behavior of equal? for user-defined classes.

(equal? (list 1 2) (list 1 2)) ⇒ #t
(equal? "abc" "abc") ⇒ #t
(equal? 100 100) ⇒ #t
(equal? 100 100.0) ⇒ #f

;; 0.0 and -0.0 is numerically equal (=), but eqv? distinguishes them, so as equal?.
(equal? 0.0 -0.0) ⇒ #f

Note: This procedure correctly handles the case when both obj1 and obj2 have cycles through pairs and vectors, as required by R6RS and R7RS. However, if the cycle involves user-defined classes, equal? may fail to terminate.

object-equal? obj1 obj2

[Generic Function] This generic function is called when equal? is called on the objects it doesn’t know about. You can define this method on your class so that equal? can check equivalence. This method is supposed to return #t if obj1 is equal to obj2, #f otherwise. If you want to check equivalence of elements recursively, do not call object-equal? directly; call equal? on each element.

(define-class <foo> ()
  ((x :init-keyword :x)
   (y :init-keyword :y)))

(define-method object-equal? ((a <foo>) (b <foo>))
  (and (equal? (slot-ref a 'x) (slot-ref b 'x))
       (equal? (slot-ref a 'y) (slot-ref b 'y))))

(equal? (make <foo> :x 1 :y (list 'a 'b))
        (make <foo> :x 1 :y (list 'a 'b)))
⇒ #t

(equal? (make <foo> :x 1 :y (make <foo> :x 3 :y 4))
        (make <foo> :x 1 :y (make <foo> :x 3 :y 4)))
⇒ #t

object-equal? (obj1 <top>) (obj2 <top>)

[Method] This method catches equal? between two objects of a user-defined class, in case the user doesn’t define a specialized method for the class.
When called, it scans the registered default comparators that can handle both \texttt{obj1} and \texttt{obj2}, and if it finds one, use the comparator’s equality predicate to see if two arguments are equal to each other. When no matching comparators are found, it just returns \texttt{#f}. See Section 6.2.4.3 [Predefined comparators], page 119, about the default comparators: Look for the entries of \texttt{default-comparator} and \texttt{comparator-register-default}.!

Note: If you define \texttt{object-equal?} with exactly the same specializers of this method, you’ll replace it and that breaks \texttt{default-comparator} operation. Future versions of Gauche will prohibit such redefinition. For now, be careful not to redefine it accidentally.

Sometimes you want to test if two aggregate structures are topologically equal, i.e., if one has a shared substructure, the other has a shared substructure in the same way. \texttt{Equal?} can’t handle it; module \texttt{util.isomorph} provides a procedure \texttt{isomorphic?} which does the job (see Section 12.84 [Determine isomorphism], page 1020).

### 6.2.2 Comparison

Equality only concern about whether two objects are equivalent or not. However, sometimes we want to see the order among objects. Again, there’s no single “universal order”. It doesn’t make mathematical sense to ask if one complex number is greater than another, but having some artificial order is useful when you want a consistent result of sorting a list of objects including numbers.

\begin{verbatim}
\textbf{compare} \texttt{obj1 obj2}                        \hspace{1cm} \textbf{[Function]}
\end{verbatim}

A general comparison procedure. Returns -1 if \texttt{obj1} is less than \texttt{obj2}, 0 if \texttt{obj1} is equal to \texttt{obj2}, and 1 if \texttt{obj1} is greater than \texttt{obj2}.

If \texttt{obj1} and \texttt{obj2} are incomparable, an error is signalled. However, \texttt{compare} defines total order between most Scheme objects, so that you can use it on wide variety of objects. The definition is upper-compatible to the order defined in SRFI-114.

Some built-in types are handled by this procedure reflecting “natural” order of comparison if any (e.g. real numbers are compared by numeric values, characters are compared by \texttt{char<} etc.) For convenience, it also defines superficial order between objects that doesn’t have natural order; complex numbers are ordered first by their real part, then their imaginary part, for example. That is, \texttt{1+i} comes before \texttt{2-i}, which comes before \texttt{2}, which comes before \texttt{2+i}.

Boolean false comes before boolean true.

Lists are ordered by dictionary order: Take the common prefix. If either one is () and the other is not, () comes first. If both tails are not empty, compare the heads of the tails. (This makes empty list the “smallest” of all lists).

Vectors (including uniform vectors) are compared first by their lengths, and if they are the same, elements are compared from left to right. Note that it’s different from lists and strings.

\begin{verbatim}
(\texttt{compare '(1 2 3) '(1 3)})
\Rightarrow -1 ; (1 2 3) is smaller
(\texttt{compare '#(1 2 3) '#(1 3)})
\Rightarrow 1 ; #(1 3) is smaller
(\texttt{compare "123" "13"})
\Rightarrow -1 ; "123" is smaller
\end{verbatim}

If two objects are of subclasses of \texttt{<object>}, a generic function \texttt{object-compare} is called.

If two objects are of different types and at least one of them isn’t \texttt{<object>}, then they are ordered by their types. SRFI-114 defines the order of builtin types as follows:

1. Empty list.
2. Pairs.
Chapter 6: Core library

5. Strings.
7. Numbers.
8. Vectors.
9. Uniform vectors (u8 < s8 < u16 < s16 < u32 < s32 < u64 < s64 < f16 < f32 < f64)
10. All other objects.

**object-compare obj1 obj2**

[Generic Function]

Specializing this generic function extends `compare` procedure for user-defined classes.

This method must return either `-1` (obj1 precedes obj2), `0` (obj1 equals to obj2), `1` (obj1 succeeds obj2), or #f (obj1 and obj2 cannot be ordered).

**object-compare (obj1 <top>) (obj2 <top>)**

[Method]

This method catches `compare` between two objects of a user-defined class, in case the user doesn’t define a specialized method for the class.

When called, it scans the registered default comparators that can handle both obj1 and obj2, and if it finds one, use the comparator’s `compare` procedure to determine the order of obj1 and obj2. When no matching comparators are found, it returns #f, meaning two objects can’t be ordered. See Section 6.2.4.3 [Predefined comparators], page 119, about the default comparators: Look for the entries of `default-comparator` and `comparator-register-default`!

Note: If you define `object-compare` with exactly the same specializers of this method, you’ll replace it and that breaks `default-comparator` operation. Future versions of Gauche will prohibit such redefinition. For now, be careful not to redefine it accidentally.

**eq-compare obj1 obj2**

[Function]

Returns -1 (less than), 0 (equal to) or 1 (greater than) according to a certain total ordering of obj1 and obj2. Both arguments can be any Scheme objects, and can be different type of objects. The following properties are guaranteed.

- (eq-compare x y) is 0 iff (eq? x y) is #t.
- The result is consistent within a single run of the process (but may differ between runs).

Other than these, no actual semantics are given to the ordering.

This procedure is useful when you need to order arbitrary Scheme objects, but you don’t care the actual order as far as it’s consistent.

### 6.2.3 Hashing

Hash functions have close relationship with equality predicate, so we list them here.

**eq-hash obj**

[Function]

These are hash functions suitable to be used with `eq?` and `eqv?`, respectively. The returned hash value is system- and process-dependent, and can’t be carried over the boundary of the running process.

Note: don’t hash numbers by `eq-hash`. Two numbers are not guaranteed to be `eq?` even if they are numerically equal.

**default-hash obj**

[R7RS+] This is a hash function suitable to be used with `equal?`. In R7RS, this is defined in `scheme.comparator` (originally in `srfi.128`).
If *obj* is either a number, a boolean, a character, a symbol, a keyword, a string, a list, a vector or a uniform vector, internal hash function is used to calculate the hash value. If *obj* is other than that, a generic function `object-hash` is called to calculate the hash value (see below).

The hash value also depends on `hash-salt`, which differs for every run of the process.

### portable-hash *obj* *salt*  
[Function]
Sometimes you need to calculate a hash value that’s “portable”, in a sense that the value won’t change across multiple runs of the process, nor between different platforms. Such hash value can be used with storing objects externally to share among processes. This procedure calculates a hash value of *obj* with such characteristics; the hash value is the same for the same object and the same salt value. Here “same object” roughly means having the same external representation. Objects `equal?` to each other are same. If you write out an object with `write`, and read it back, they are also the same objects in this sense. This means objects without read/write invariance, such as ports, can’t be handled with `portable-hash`. It is caller’s responsibility that *obj* won’t contain such objects.

The *salt* argument is a nonnegative fixnum and gives variations in the hash function. You have to use the same salt to get consistent results.

If *obj* is other than a number, a boolean, a character, a symbol, a keyword, a string, a list, a vector, or a uniform vector, this procedure calls a generic function `object-hash` is called to calculate the hash value (see below).

### legacy-hash *obj*  
[Function]
Up to 0.9.4, Gauche had a hash function called `hash` that was used in both `equal?`-hashtable and for the portable hash function. It had a problem, though.

1. There was no way to salt the hash function, which makes the hashtables storing externally provided data vulnerable to collision attack.
2. The hash function behaves poorly, especially on flonums.
3. There are bugs in bignum and flonum hashing code that have produced different results on different architectures.

Since there are existing hash values calculated with the old hash function, we preserve the behavior of the original `hash` function as `legacy-hash`. Use this when you need to access old data. (The `hash` function also behaves as `legacy-hash` by default, but it has tweak; see below.)

The new code that needs portable hash value should use `portable-hash` instead.

### object-hash *obj* *rec-hash*  
[Generic Function]
By defining a method for this generic function, objects of user-defined types can have a hash value and can be used in a `equal?` hash table.

The method has to return an exact non-negative integer, and must return the same value for two object which are `equal?`. Furthermore, the returned value must not rely on the platform or state of the process, if *obj* is a portable object (see `portable-hash` above for what is portable.)

If the method needs to get hash value of *obj*’s elements, it has to call `rec-hash` on them. It guarantees that the proper hash function is called recursively. So you can count on `rec-hash` to calculate a portable hash value when `object-hash` itself is called from `portable-hash`.

If *obj* has several elements, you can call `combine-hash-value` on the elements’ hash values.

```scheme
(define-class <myclass> () (x y))
```
;; user-defined equality function
(define-method object-equal? ((a <myclass>) (b <myclass>))
  (and (equal? (ref a 'x) (ref b 'x))
       (= (abs (ref a 'y)) (abs (ref b 'y))))

;; user-defined hash function
(define-method object-hash ((a <myclass>) rec-hash)
  (combine-hash-value (rec-hash (ref a 'x))
                       (rec-hash (abs (ref a 'y))))

Note: The base method of object-hash hashes any object to a single hash value (the actual value depends on hash-salt if object-hash is called from default-hash, and a fixed constant value otherwise. It’s because object’s equality semantics can be customized separately, and we can’t compute a non-constant hash value without knowing the equality semantics. This behavior is the last “safety net”; in general, you should define object-hash method on your class if the instances of your class can ever be hashed.

object-hash (obj <top>) rec-hash \[Method\]
object-hash (obj <top>) \[Method\]
  These two methods are defined by the system and ensures the backward compatibility and the behavior of default-comparator. Be careful not to replace these methods by defining the exactly same specializers. In future versions of Gauche, attempts to replace these methods will raise an error.

combine-hash-value ha hb \[Function\]
  Returns a hash value which is a combination of two hash values, ha and hb. The guaranteed invariance is that if (= ha1 ha2) and (= hb1 hb2) then (= (combine-hash-value ha1 hb1) (combine-hash-value ha2 hb2)). This is useful to write user-defined object-hash method.

hash obj \[Function\]
  This function is deprecated.
  Calculate a hash value of obj suitable for equal? hash. By default, it returns the same value as legacy-hash. However, if this is called from default-hash or portable-hash (via object-hash method), it recurses to the calling hash function.
  The behavior is to keep the legacy code work. Until 0.9.5, hash is the only hash function to be used for both portable hash and equal?-hash, and object-hash method takes single argument (an object to hash) and calls hash recursively whenever it needs to get a hash value of other objects pointed from the argument.
  As of 0.9.5 we have more than one hash functions that calls object-hash, so the method takes the hash function as the second argument to recurse. However, we can’t just break the legacy code; so there’s a default method defined in object-hash which is invoked when no two-arg method is defined for the given object, and dispatches to one-arg method. As far as the legacy object-hash code calls hash, it calls proper function. The new code shouldn’t rely on this behavior, and must use the second argument of object-hash instead.

boolean-hash bool \[Function\]
char-hash char \[Function\]
char-ci-hash char \[Function\]
string-hash str \[Function\]
string-ci-hash str \[Function\]
symbol-hash sym \[Function\]
number-hash num \[Function\]
[R7RS comparator] These are hash functions for specific type of objects, defined in R7RS scheme.comparator. In Gauche, these procedures are just a wrapper of default-hash.
with type checks (and case folding when relevant). These are mainly provided to con-
form scheme.comparator; in your code you might just want to use default-hash (or
eq-hash/eqv-hash, depending on the equality predicate).

The case-folding versions, char-ci-hash and string-ci-hash, calls char-foldcase and
string-foldcase respectively, on the argument before passing it to hash. (See Section 6.9
[Characters], page 160, for char-foldcase. See Section 9.37.3 [Full string case conversion],
page 554, for string-foldcase).

hash-bound                     [Function]
hash-salt                      [Function]

[R7RS comparator] Both evaluates to an exact nonnegative integers. In R7RS, these are
defined in scheme.comparator.

(Note: scheme.comparator defines these as macros, in order to allow implementations opti-
mize runtime overhead. In Gauche we use procedures but the overhead is negligible.)

User-defined hash functions can limit the range of the result between 0 and (hash-bound),
respectively, without worrying to lose quality of hash function. (User-defined hash functions
don’t need to honor (hash-bound) at all; hashtables takes modulo when necessary.)

User-defined hash function can also take into account of the value (hash-salt) into hash
calculation; the salt value may differ between runs of the Scheme processes, or even between
hash table instances. It is to avoid collision attack. Built-in hash functions already takes the
salt value into account, so if your hash function is combining the hash values of primitive
types, you don’t need to worry about salt values.

6.2.4 Basic comparators

Equality and comparison procedures are parameters in various data structures. A treemap needs
to order its keys; a hashtable needs to see if the keys are the same or not, and it also need a
hash function consistent with the equality predicate.

If we want to work on generic data structures, we need to abstract those variations of compar-
ison schemes. So here comes the comparator, a record that bundles closely-related comparison
procedures together.

There are two SRFIs that define comparators. The one that was originally called SRFI-128
has now become a part of R7RS large as scheme.comparator, and we recommend new code to
use it. Gauche has all of scheme.comparator procedures built-in. The older, and rather complex
one is SRFI-114; Gauche also supports it mainly for the backward compatibility. Importantly,
Gauche’s native <comparator> object is compatible to both scheme.comparator and srfi-114
comparators.

6.2.4.1 Comparator class and constructors

<comparator>                    [Builtin Class]

A comparator record that bundles the following procedures:

Type test predicate
Checks if an object can be compared with this comparator.

Equality predicate
See if given two objects are equal to each other; returns a boolean value.

Ordering predicate
Compare given two objects, and returns true iff the first one is strictly precedes
the second one. That is, this is a less-than predicate.
Comparison procedure

Compare given two objects, and returns either -1 (the first one is less than the second), 0 (they are equal), or 1 (the first one is greater than the second).

Hash function

Returns a hash value of the given object.

Scheme.comparator's comparators use the ordering predicate, while SRFI-114 comparators use the comparison procedure. Gauche's <comparator> supports both by automatically generating the missing one; that is, if you create a comparator with scheme.comparator interface, by giving an ordering predicate, Gauche automatically fills the comparison procedure, and if you create one with SRFI-114 interface by giving a comparison procedure, Gauche generates the ordering predicate.

A comparator may not have an ordering predicate / comparison procedure, and/or a hash function. You can check if the comparator can be used for ordering or hashing by comparator-ordered? and comparator-hashable?, respectively.

Some built-in data types such as hashtables (see Section 6.14.1 [Hashtables], page 206) and treemaps (see Section 6.14.2 [Treemaps], page 212), take a comparator in their constructors. The sort and merge procedures also accept comparators (see Section 6.23 [Sorting and merging], page 291).

make-comparator  type-test  equal  order  hash  :optional  name
[Function] [R7RS comparator] Creates a new comparator form the given type-test, equal, order and hash functions, and returns it. In R7RS, this is defined in scheme.comparator

See the description of <comparator> above for the role of those procedures.

Note: Both scheme.comparator and srfi.114 defines make-comparator, but where scheme.comparator takes order argument, srfi.114 takes compare argument. Since scheme.comparator is preferable, we adopt it for the built-in interface, and give a different name (make-comparator/compare) for SRFI-114 constructor.

Actually, some arguments can be non-procedures, to use predefined procedures, for the convenience. Even if non-procedure arguments are passed, the corresponding accessors (e.g. comparator-type-test-procedure for the type-test procedure) always return a procedure—either the given one or the predefined one.

The type-test argument must be either #t or a predicate taking one argument to test suitability of the object for comparing by the resulting comparator. If it is #t, a procedure that always return #t is used.

The equal argument must a predicate taking two arguments to test equality.

The order argument must be either #f or a procedure taking two arguments and returning a boolean value. It must return #t iff the first argument strictly precedes the second one. If #f is passed, the comparator can not be used for ordering.

The hash argument must be either #f, or a procedure taking one argument and returning nonnegative exact integer. If #f is given, it indicates the comparator can't hash objects; the predefined procedure just throws an error.

The fifth, optional argument name, is Gauche's extension. It can be any object but usually a symbol; it is only used when printing the comparator, to help debugging.

make-comparator/compare  type-test  equal  compare  hash  :optional  name
[Function] This is SRFI-114 comparator constructor. In SRFI-114, this is called make-comparator. Avoiding name conflict, we renamed it. If you (use srfi.114) you get the original name make-comparator (and the built-in make-comparator is shadowed). This is provided for the backward compatibility, and new code should use built-in make-comparator above.
It’s mostly the same as `make-comparator` above, except the following:

- The third argument (compare) is a comparison procedure instead of an ordering predicate. It must be either `#f`, or a procedure taking two arguments and returning either -1, 0, or 1, depending on whether the first argument is less than, equal to, or greater than the second argument. If it is `#f`, it indicates the comparator can’t order objects.
- You can pass `#t` to the `equal` argument when you give a comparison procedure. In that case, equality is determined by calling the comparison procedure and see if the result is 0.

### 6.2.4.2 Comparator predicates and accessors

```scheme
comparator? obj

[R7RS comparator] Returns true iff obj is a comparator. In R7RS, this is provided from `scheme.comparator`.
```

```scheme
object-equal? (a <comparator>) (b <comparator>)

[Method] Comparing two comparators by `equal?` compares their contents, via this method. Even `a` and `b` are comparators created separately, they can be `equal?` if all of their slots are the same.
```

This is Gauche’s extension. The standard says nothing about equality of comparators, but it is sometimes useful if you can compare two.

```scheme
(equal? (make-comparator #t equal? #f hash ’foo) (make-comparator #t equal? #f hash ’foo))
⇒ #t
```

```scheme
;; The following may be #t or #f, depending on how the anonymous ;; procedure is allocated.
(equal? (make-comparator (^x x) eq? #f #f) (make-comparator (^x x) eq? #f #f))
```

```scheme
comparator-flavor cmpr

[Function] Returns a symbol `ordering` if `cmpr` is created with `scheme.comparator` constructor, and returns `comparison` if `cmpr` is created with SRFI-114 constructor.
```

Usually applications don’t need to distinguish these two kinds of comparators, for either kind of comparators can behave just as another kind. This procedure is for some particular cases when one wants to optimize for the underlying comparator implementation.

```scheme
comparator-ordered? cmpr
comparator-hashable? cmpr

[R7RS comparator] Returns true iff a comparator `cmpr` can be used to order objects, or to hash them, respectively. In R7RS, this is provided from `scheme.comparator`.
```

```scheme
comparator-type-test-procedure cmpr
comparator-equality-predicate cmpr
comparator-ordering-predicate cmpr
comparator-hash-function cmpr

[R7RS comparator] Returns type test procedure, equality predicate, ordering procedure and hash function of comparator `cmpr`, respectively. In R7RS, this is provided from `scheme.comparator`.
```

These accessors always return procedures; if you give `#f` to the `order` or `hash` argument of the constructor, `comparator-ordering-predicate` and `comparator-hash-function` still return a procedure, which will just raise an error.
comparator-comparison-procedure \texttt{cmpr} \hspace{1cm} \textbf{[Function]}

[SRFI-114] This is a SRFI-114 procedure, but sometimes handy with \texttt{scheme.comparator} comparators. Returns a procedure that takes two objects that satisfy the type predicates of \texttt{cmpr}. The procedure returns either \(-1\), \(0\) or \(1\), depending on whether the first object is less than, equal to, or greater than the second. The comparator must be ordered, that is, it must have an ordering predicate (or a comparison procedure, if it is created by SRFI-114 constructor).

comparator-test-type \texttt{cmpr} \texttt{obj} \hspace{1cm} \textbf{[Function]}
comparator-check-type \texttt{cmpr} \texttt{obj} \hspace{1cm} \textbf{[Function]}

[R7RS comparator] Test whether \texttt{obj} can be handled by a comparator \texttt{cmpr}, by applying \texttt{cmpr}'s type test predicate. The former (\texttt{comparator-test-type}) returns a boolean value, while the latter (\texttt{comparator-check-type}) signals an error when \texttt{obj} can't be handled.

In R7RS, this is provided from \texttt{scheme.comparator}.

=? \texttt{cmpr} \texttt{obj} \texttt{obj2} \texttt{obj3} \ldots \hspace{1cm} \textbf{[Function]}
<? \texttt{cmpr} \texttt{obj} \texttt{obj2} \texttt{obj3} \ldots \hspace{1cm} \textbf{[Function]}
<=? \texttt{cmpr} \texttt{obj} \texttt{obj2} \texttt{obj3} \ldots \hspace{1cm} \textbf{[Function]}
>? \texttt{cmpr} \texttt{obj} \texttt{obj2} \texttt{obj3} \ldots \hspace{1cm} \textbf{[Function]}
>=? \texttt{cmpr} \texttt{obj} \texttt{obj2} \texttt{obj3} \ldots \hspace{1cm} \textbf{[Function]}

[R7RS comparator] Compare objects using a comparator \texttt{cmpr}. All of \texttt{obj}, \texttt{obj2}, \texttt{obj3} \ldots must satisfy the type predicate of \texttt{cmpr}. When more than two objects are given, the order of comparison is undefined.

In order to use <?, <=?, >? and >=?, comparator must be ordered.

In R7RS, this is provided from \texttt{scheme.comparator}.

comparator-hash \texttt{cmpr} \texttt{obj} \hspace{1cm} \textbf{[Function]}

[R7RS comparator] Returns a hash value of \texttt{obj} with the hash function of a comparator \texttt{cmpr}. The comparator must be hashable, and \texttt{obj} must satisfy comparator’s type test predicate.

In R7RS, this is provided from \texttt{scheme.comparator}.

comparator-compare \texttt{cmpr} \texttt{a} \texttt{b} \hspace{1cm} \textbf{[Function]}

[SRFI-114] Order two objects \texttt{a} and \texttt{b} using \texttt{cmpr}, and returns either one of \(-1\) (\texttt{a} is less than \texttt{b}), \(0\) (\texttt{a} equals to \texttt{b}), or \(1\) (\texttt{a} is greater than \texttt{b}). Objects must satisfy \texttt{cmpr}'s type test predicate.

A simple comparison can be done by <? etc, but sometimes three-way comparison comes handy. So we adopt this procedure from SRFI-114.

6.2.4.3 Predefined comparators

default-comparator \hspace{1cm} \textbf{[Variable]}

[SRFI-114] This variable bounds to a comparator that is used by default in many context.

It can compare most of Scheme objects, even between objects with different types. In fact, it is defined as follows:

\begin{verbatim}
(define default-comparator
  (make-comparator/compare #t equal? compare default-hash 'default-comparator))
\end{verbatim}

As you see in the definition, equality, ordering and hashing are handled by \texttt{equal?}, \texttt{compare} and \texttt{default-hash}, respectively. They takes care of built-in objects, and also \texttt{equal?} and \texttt{compare} handle the case when two objects of different types.
For objects of user-defined classes, those procedures call generic functions \texttt{object-equal?}, \texttt{object-compare}, and \texttt{object-hash}, respectively. Defining methods for them automatically extended the domain of \texttt{default-comparator}.

\texttt{Scheme.comparator} defines another way to extend \texttt{default-comparator}. See \texttt{comparator-register-default!} below for the details.

\texttt{comparator-register-default! \texttt{comparator}} \hfill \textbf{[Function]}

\texttt{[R7RS comparator]} In R7RS, this is provided from \texttt{scheme.comparator}. This is the \texttt{scheme.comparator} way for user programs to extend the behavior of the \texttt{default-comparator} (which is what \texttt{make-default-comparator} returns).

Note that, in Gauche, you can also extend default comparator’s behavior by defining specialized methods for \texttt{object-equal?}, \texttt{object-compare} and \texttt{object-hash}. See the description of \texttt{default-comparator} above, for the details.

In fact, Gauche uses those generic functions to handle the registered comparators; methods specialized for \texttt{<top>} are defined for those generic functions, which catches the case when \texttt{default-comparator} is applied on object(s) of user-defined classes that don’t have specialized methods defined for those generic functions. The catching method examines registered comparators to find one that can handle passed argument(s), and if it finds one, use it.

You might frown at this procedure having a global side-effect. Well, \texttt{scheme.comparator} explicitly prohibits comparators registered by this procedure alters the behavior of the default comparator in the existing domain—it is only allowed to handle objects that aren’t already handled by the system’s original default comparator and other already registered comparators. So, the only effect of adding new comparator should make the default comparator work on objects that had been previously raised an error.

In reality, it is impossible to enforce the condition. If you register a comparator whose domain overlaps the domain the default comparator (and its extensions via Gauche’s methods), the program becomes non-portable at that moment. In the current version, the comparators registered by \texttt{comparator-register-default!} has the lowest precedence on the dispatch mechanism, but you shouldn’t count on that.

\texttt{eq-comparator} \hfill \textbf{[Variable]}
\texttt{eqv-comparator} \hfill \textbf{[Variable]}
\texttt{equal-comparator} \hfill \textbf{[Variable]}

\texttt{[SRFI-114]} Built-in comparators that uses \texttt{eq?}, \texttt{eqv?} and \texttt{equal?} for the equality predicate, respectively. They accept any kind of Scheme objects. Each has corresponding hash functions (i.e. \texttt{eq-hash} for \texttt{eq-comparator}, \texttt{eqv-hash} for \texttt{eqv-comparator} and \texttt{default-hash} for \texttt{equal-comparator}). Only \texttt{eq-comparator} is ordered, using \texttt{eq-compare} to order the objects (see Section 6.2.2 [Comparison], page 112, for \texttt{eq-compare}).

Note that \texttt{eq-comparator} and \texttt{eqv-comparator} are not equivalent from what \texttt{make-eq-comparator} and \texttt{make-eqv-comparator} return, respectively. The latter two are defined in \texttt{scheme.comparator} and specified to use \texttt{default-hash} for the hash function. It is heavier than \texttt{eq-hash/eqv-hash}, and it can’t be used for circular objects, nor for the mutable objects with which you want to hash them by identity. We provide \texttt{eq-comparator} and \texttt{eqv-comparator} in case you want to avoid limitations of \texttt{default-hash}.

\texttt{boolean-comparator} \hfill \textbf{[Variable]}
\texttt{char-comparator} \hfill \textbf{[Variable]}
\texttt{char-ci-comparator} \hfill \textbf{[Variable]}
\texttt{string-comparator} \hfill \textbf{[Variable]}
\texttt{string-ci-comparator} \hfill \textbf{[Variable]}

\texttt{[SRFI-114]} Compare booleans, characters, and strings, respectively. The \texttt{*-ci-*} variants uses case-insensitive comparison. All have appropriate hash functions, too.
The string case-insensitive comparison uses Unicode full-string case conversion (see Section 9.37.3 [Full string case conversion], page 554).

The string case-insensitive comparison uses Unicode full-string case conversion (see Section 9.37.3 [Full string case conversion], page 554).

### predefined comparators

- **exact-integer-comparator**
- **integer-comparator**
- **rational-comparator**
- **real-comparator**
- **complex-comparator**
- **number-comparator**

  [SRFI-114] Compare exact integers, integers, rational numbers, real numbers, complex numbers and general numbers, respectively. In Gauche **number-comparator** is the same as **complex-comparator**.

  The equality are determined by `=`. For exact integer, integer, rational and real comparators, the order is the numerical order. Two complex numbers are compared first by their real components, and then their imaginary components only if the real components are the same.

  Note that those comparator rejects NaN. You need **make-inexact-real-comparator** in **srfi.114** module to compare NaNs with your own discretion. See Section 11.24 [Comparators], page 731, for the details.

- **pair-comparator**
- **list-comparator**
- **vector-comparator**
- **uvector-comparator**
- **bytevector-comparator**

  [SRFI-114] The default comparators to compare pairs, lists, vectors, uniform vectors and bytevectors (which is synonym to u8vector). Their respective elements are compared with the default comparators.

  Note that lists are compared by dictionary order ((1 2 3) comes before (1 3)), while in vector-families shorter ones are ordered first (#(1 3) comes before #(1 2 3)).

- **time-comparator**

  A comparator to compare `<time>` objects (see Section 6.24.9 [Time], page 316).

### Combining comparators

- **make-default-comparator**

  [R7RS comparator] Returns a default comparator. In Gauche, this returns the **default-comparator** object. In R7RS, this is provided from **scheme.comparator**.

- **make-eq-comparator**
- **make-eqv-comparator**

  [R7RS comparator] Returns comparators that use `eq?` and `eqv?` for its equality predicate, respectively. Note that they use **default-hash** for hash functions, as specified by **scheme.comparator**, which has a few drawbacks: You can’t use it if you want to hash based on identity of mutable objects, it diverges on circular objects, and it is slow if applied on a large structures. We recommend to use **eq-comparator** or **eqv-comparator** if possible (see Section 6.2.4.3 [Predefined comparators], page 119).

  In R7RS, this is provided from **scheme.comparator**.

- **make-reverse-comparator**

  [SRFI-114] Returns a comparator with the same type test predicate, equality procedure, and hash function as the given comparator, but the comparison procedure is flipped.
**make-key-comparator cmpr test key**  
[Function]  
Suppose you have some kind of structure, but you only need to look at one part of it to compare them.

Returns a new comparator that uses test as type test predicate. Its equality predicate, comparison procedure and hash function are constructed by applying key to the argument(s) then passing the result to the corresponding procedure of cmpr. If cmpr lacks comparison procedure and/or hash function, so does the returned comparator.

In the following example, the tree-map users compares the given user records only by the username slots:

```scheme
(use gauche.record)

(define-record-type user #t #t
    username ; string
    password-hash ; string
    comment) ; string

(define users ; table of users, managed by tree-map
    (make-tree-map
        (make-key-comparator string-comparator user? user-username)))
```

See Section 11.57 [Composing comparators], page 801, for additional procedures to create comparator of aggregate types.

**make-tuple-comparator cmpr1 cmpr2 ...**  
[Function]  
Creates a comparator that compares lists of the form \((x_1 x_2 \ldots)\), where each element is compared with the corresponding comparator. For example, \((\text{make-tuple-comparator } c1 c2 c3)\) will compare three-element list, whose first elements are compared by \(c1\), second elements by \(c2\) and third elements by \(c3\).

See Section 11.57 [Composing comparators], page 801, for additional procedures to create comparator of aggregate types.

### 6.3 Numbers

Gauche supports the following types of numbers

- multi-precision exact integer
  There’s no limit of the size of number except the memory of the machine.

- multi-precision exact non-integral rational numbers.
  Both denominator and numerator are represented by exact integers. There’s no limit of the size of number except the memory of the machine.

- inexact floating-point real numbers
  Using double-type of underlying C compiler, usually IEEE 64-bit floating point number.

- inexact floating-point complex numbers
  Real part and imaginary part are represented by inexact floating-point real numbers.

#### 6.3.1 Number classes

- `<number>` [Builtin Class]
- `<complex>` [Builtin Class]
- `<real>` [Builtin Class]
- `<rational>` [Builtin Class]
Chapter 6: Core library

mediatek

6.3.2 Numerical predicates

number? obj [Function]
complex? obj [Function]
real? obj [Function]
rational? obj [Function]
integer? obj [Function]

[R7RS base] Returns #t if obj is a number, a complex number, a real number, a rational number or an integer, respectively. In Gauche, a set of numbers is the same as a set of complex numbers. A set of rational numbers is the same as a set of real numbers, except +inf.0, -inf.0 and +nan.0 (since we have only limited-precision floating numbers).

(real? 3) ⇒ #t
(real? -2.5+0.0i) ⇒ #t
(integer? 3+0i) ⇒ #t
(integer? 3.0) ⇒ #t

(real? +inf.0) ⇒ #t
(real? +nan.0) ⇒ #t

(rational? +inf.0) ⇒ #f
(rational? +nan.0) ⇒ #f

Note: R6RS adopts more strict definition on exactness, and notably, it defines a complex number with non-exact zero imaginary part is not a real number. Currently Gauche doesn’t have exact complex numbers, and automatically coerces complex numbers with zero imaginary part to a real number. Thus R6RS code that relies on the fact that (real? 1+0.0i) is #f won’t work with Gauche.

real-valued? obj [Function]
rational-valued? obj [Function]
integer-valued? obj [Function]

[R6RS] In Gauche these are just an alias of real?, rational? and integer?. They are provided for R6RS compatibility.
The difference of those and non-valued versions in R6RS is that these returns #t if \texttt{obj} is a complex number with nonexact zero imaginary part. Since Gauche doesn't distinguish complex numbers with zero imaginary part and real numbers, we don't have the difference.

\begin{verbatim}
exact? \texttt{obj} | [Function]
inexact? \texttt{obj} | [Function]
\textbf{[R7RS base]} Returns #t if \texttt{obj} is an exact number and an inexact number, respectively.
\hspace{1em} (exact? 1) \Rightarrow #t
\hspace{1em} (exact? 1.0) \Rightarrow #f
\hspace{1em} (inexact? 1) \Rightarrow #f
\hspace{1em} (inexact? 1.0) \Rightarrow #t
\hspace{1em} (exact? (modulo 5 3)) \Rightarrow #t
\hspace{1em} (inexact? (modulo 5 3.0)) \Rightarrow #f
\end{verbatim}

\begin{verbatim}
extact-integer? \texttt{obj} | [Function]
\textbf{[R7RS base]} Same as (and (exact? \texttt{obj}) (integer? \texttt{obj})), but more efficient.
\end{verbatim}

\begin{verbatim}
zero? \texttt{z} | [Function]
\textbf{[R7RS base]} Returns #t if a number \texttt{z} equals to zero.
\hspace{1em} (zero? 1) \Rightarrow #f
\hspace{1em} (zero? 0) \Rightarrow #t
\hspace{1em} (zero? 0.0) \Rightarrow #t
\hspace{1em} (zero? -0.0) \Rightarrow #t
\hspace{1em} (zero? 0.0+0.0i) \Rightarrow #t
\end{verbatim}

\begin{verbatim}
positive? \texttt{x} | [Function]
negative? \texttt{x} | [Function]
\textbf{[R7RS base]} Returns #t if a real number \texttt{x} is positive and negative, respectively. It is an error to pass a non-real number.
\hspace{1em} Note that minus zero (-0.0) is zero, not negative.
\end{verbatim}

\begin{verbatim}
negative-zero? \texttt{x} | [Function]
\textbf{Returns #t if a number \texttt{x} is a minus zero (-0.0), #f otherwise. An error is signaled if \texttt{x} is not a number.}
\end{verbatim}

\begin{verbatim}
finite? \texttt{z} | [Function]
infinite? \texttt{z} | [Function]
nan? \texttt{z} | [Function]
\textbf{[R7RS inexact]} For real numbers, returns #f iff the given number is finite, infinite, or NaN, respectively.
\hspace{1em} For non-real complex numbers, \texttt{finite?} returns #t iff both real and imaginary components are finite, \texttt{infinite?} returns #t if at least either real or imaginary component is infinite, and \texttt{nan?} returns #t if at least either real or imaginary component is NaN. (Note: It is incompatible to R6RS, in which these procedures must raise an error if the given argument is non-real number.)
\hspace{1em} In R7RS, these procedures are in \texttt{(scheme inexact)} library.
\end{verbatim}

\begin{verbatim}
odd? \texttt{n} | [Function]
even? \texttt{n} | [Function]
\textbf{[R7RS base]} Returns #t if an integer \texttt{n} is odd and even, respectively. It is an error to pass a non-integral number.
\hspace{1em} (odd? 3) \Rightarrow #t
\end{verbatim}
(even? 3)  ⇒  #f
(odd? 3.0) ⇒ #t

fixnum? n  [Function]
bignum? n  [Function]
  [R7RS fixnum] Returns #t if n is an exact integer whose internal representation is fixnum
  and bignum, respectively. R7RS-large defines fixnum? in scheme.fixnum library; bignum? is
  Gauche’s extension. Portable Scheme programs don’t need to care about the internal repre-
  sentation of integer. These are for certain low-level routines that does particular optimization.
  See Section 10.3.23 [R7RS fixnum], page 668, for the comprehensive fixnum library.

flonum? x  [Function]
  [R7RS flonum] Returns #t if x is a number represented by a floating-point number, #f
  otherwise. In Gauche, inexact real numbers are flonums.
  See Section 10.3.24 [R7RS flonum], page 670, for comprehensive flonum library.

ratnum? x  [Function]
  Returns #t if x is a number represented by an exact non-integral number, or #f otherwise.
  Internally, a number that returns #t for it is represented as a pair of two exact integers.
  Usually you don’t need to distinguish ratnums from exact integers; you can treat them as
  exact numbers. In performance-sensitive code, however, ratnum slows down computation a
  lot and you may want to detect that case.

6.3.3 Numerical comparison

=  z1 z2 z3 ...  [Function]
  [R7RS base] If all the numbers z are equal numerically, returns #t.
  (= 2 2)    ⇒  #t
  (= 2 3)    ⇒  #f
  (= 2/4 1/2) ⇒ #t

  Exactness doesn’t affect numerical comparison; inexact 1.0 and exact 1 are = to each other.
  Positive inexact zero (0.0) and negative inexact zero (-0.0) are also = to each other. To
  distinguish numerically equal exact and inexact number, you have to use eqv? or equal?.
  (= 2 2.0)    ⇒  #t
  (= 2.0 2.0+0i) ⇒ #t
  (= -0.0 0.0) ⇒ #t

  ;; cf:
  (eqv? 2 2.0)    ⇒  #f
  (eqv? -0.0 0.0) ⇒ #f

  Note that +nan.0 would never = to any number, including itself.
  (= +nan.0 +inf.0) ⇒ #f
  (= +nan.0 +nan.0) ⇒ #f

  (let ((x +nan.0)) (= x x)) ⇒ #f

< x1 x2 x3 ...  [Function]
<= x1 x2 x3 ...  [Function]
> x1 x2 x3 ...  [Function]
>= x1 x2 x3 ...  [Function]
  [R7RS base] Returns #t If all the real numbers x are monotonically increasing, monotonically
    nondecreasing, monotonically decreasing, or monotonically nonincreasing, respectively.
Since \((= 0.0 -0.0)\) is \#t, \((> 0.0 -0.0)\) is \#f.

If any of the argument is NaN, the result is always \#f. Hence \((< x y) \Rightarrow \#f\) does not imply \((\geq x y) \Rightarrow \#t\).

**max** \(x_1 x_2 \ldots\)  
**min** \(x_1 x_2 \ldots\)

[Function]  

[R7RS base] Returns a maximum or minimum number in the given real numbers, respectively. If any of the arguments are NaN, NaN is returned.

See also **find-min** and **find-max** in Section 9.6.2 [Selection and searching in collection], page 408.

**min\&max** \(x_1 x_2 \ldots\)

Returns a maximum and minimum number in the given real numbers.  
See also **find-min\&max** in Section 9.6.2 [Selection and searching in collection], page 408.

**approx=?** \(x y\) :optional relative-tolerance absolute-tolerance  
[Function]  

Returns \#t iff two numbers are approximately equal within the given error tolerance.

- If at least one of \(x\) or \(y\) is NaN, returns \#f.
- If either one is infinity, returns \#t iff the other one is also infinity of the same sign.
- Otherwise, return a boolean value computed as follows:

\[
(<= (abs (- x y))
  (max (* (max (abs x) (abs y)) relative-tolerance)
    absolute-tolerance))
\]

If at least one of \(x\) or \(y\) are non-real complex number, **magnitude** is used in place of **abs**.

When omitted, **relative-tolerance** is assumed to be \((flonum-epsilon)\), and **absolute-tolerance** is \((flonum-min-denormalized)\). That is, by default, **approx=?** tolerates 1 ULP (unit in the last place) error.

The **absolute-tolerance** argument is useful when arguments are close to zero, in which case relative tolerance becomes too small.

**flonum-epsilon**  
**flonum-min-normalized**  
**flonum-min-denormalized**  

Returns flonums with the following characteristics, respectively:

- **flonum-epsilon**
  Returns the least positive flonum \(e\) such that, for a normalized flonum \(x\), \(x\) and \((* x (+ 1.0 e))\) are distinguishable.

- **flonum-min-normalized**
  Returns the least positive flonum representable as normalized floating-point number.

- **flonum-min-denormalized**
  Returns the least positive flonum representable as denormalized floating-point number. If the platform doesn’t support denormalized flonum, it returns the least positive normalized floating number.

### 6.3.4 Arithmetics

**+** \(z \ldots\)  
**\(*\)** \(z \ldots\)

[Function]  

[R7RS base] Returns the sum or the product of given numbers, respectively. If no argument is given, \((+)\) yields 0 and \((\*)\) yields 1.
- \( z_1 \ z_2 \ldots \) [Function]

/ \( z_1 \ z_2 \ldots \) [Function]

[R7RS base] If only one number \( z_1 \) is given, returns its negation and reciprocal, respectively.

If more than one number are given, returns:

\[
\begin{align*}
   z_1 - z_2 - z_3 \ldots \\
   z_1 / z_2 / z_3 \ldots 
\end{align*}
\]

respectively.

\[
\begin{align*}
   (- 3) & \Rightarrow -3 \\
   (- -3.0) & \Rightarrow 3.0 \\
   (- 5+2i) & \Rightarrow -5.0-2.0i \\
   (/ 3) & \Rightarrow 1/3 \\
   (/ 5+2i) & \Rightarrow 0.172413793103448-0.0689655172413793i \\
   (- 5 \ 2 \ 1) & \Rightarrow 2 \\
   (- 5 \ 2.0 \ 1) & \Rightarrow 2.0 \\
   (- 5+3i \ -i) & \Rightarrow 5.0+2.0i \\
   (/ \ 14 \ 6) & \Rightarrow 7/3 \\
   (/ \ 6+2i \ 2) & \Rightarrow 3.0+1.0i 
\end{align*}
\]

Note: Gauche didn’t have exact rational number support until 0.8.8; before that, \( / \) coerced the result to inexact even if both divisor and dividend were exact numbers, when the result wasn’t a whole number. It is not the case anymore.

If the existing code relies on the old behavior, it runs very slowly on the newer versions of Gauche, since the calculation proceeds with exact rational arithmetics that is much slower than floating point arithmetics. You want to use \( / \). below to use fast inexact arithmetics (unless you need exact results).

+. \( z \ldots \) [Function]

*. \( z \ldots \) [Function]

-. \( z_1 \ z_2 \ldots \) [Function]

/. \( z_1 \ z_2 \ldots \) [Function]

Like +, *, -, and /, but the arguments are coerced to inexact number. So they always return inexact number. These are useful when you know you don’t need exact calculation and want to avoid accidental overhead of bignums and/or exact rational numbers.

abs \( z \) [Function]

[R7RS+] For real number \( z \), returns an absolute value of it. For complex number \( z \), returns the magnitude of the number. The complex part is Gauche extension.

\[
\begin{align*}
   (\text{abs} \ -1) & \Rightarrow 1 \\
   (\text{abs} \ -1.0) & \Rightarrow 1.0 \\
   (\text{abs} \ 1+i) & \Rightarrow 1.4142135623731 
\end{align*}
\]

quotient \( n1 \ n2 \) [Function]

remainder \( n1 \ n2 \) [Function]

modulo \( n1 \ n2 \) [Function]

[R7RS base] Returns the quotient, remainder and modulo of dividing an integer \( n1 \) by an integer \( n2 \). The result is an exact number only if both \( n1 \) and \( n2 \) are exact numbers.

Remainder and modulo differ when either one of the arguments is negative. Remainder \( R \) and quotient \( Q \) have the following relationship.

\[
n1 = Q \ast n2 + R
\]

where \( \text{abs}(Q) = \text{floor}(\text{abs}(n1)/\text{abs}(n2)) \). Consequently, \( R \)'s sign is always the same as \( n1 \)'s.
On the other hand, modulo works as expected for positive \( n_2 \), regardless of the sign of \( n_1 \) (e.g. \( \text{modulo} -1 \, n_2 \equiv n_2 - 1 \)). If \( n_2 \) is negative, it is mapped to the positive case by the following relationship.

\[
\text{modulo}(n_1, n_2) = -\text{modulo}(-n_1, -n_2)
\]

Consequently, modulo’s sign is always the same as \( n_2 \)’s.

\[
\begin{align*}
(\text{remainder} \ 10 \ 3) & \Rightarrow 1 \\
(\text{modulo} \ 10 \ 3) & \Rightarrow 1 \\
(\text{remainder} \ -10 \ 3) & \Rightarrow -1 \\
(\text{modulo} \ -10 \ 3) & \Rightarrow 2 \\
(\text{remainder} \ 10 \ -3) & \Rightarrow 1 \\
(\text{modulo} \ 10 \ -3) & \Rightarrow -2 \\
(\text{remainder} \ -10 \ -3) & \Rightarrow -1 \\
(\text{modulo} \ -10 \ -3) & \Rightarrow -1
\end{align*}
\]

\( \text{quotient}\&\text{remainder} \ n_1 \ n_2 \) \ [Function]

Calculates the quotient and the remainder of dividing integer \( n_1 \) by integer \( n_2 \) simultaneously, and returns them as two values.

\[
\begin{align*}
\text{div} \ x \ y & \quad \text{[Function]} \\
\text{mod} \ x \ y & \quad \text{[Function]} \\
\text{div-and-mod} \ x \ y & \quad \text{[Function]} \\
\text{div0} \ x \ y & \quad \text{[Function]} \\
\text{mod0} \ x \ y & \quad \text{[Function]} \\
\text{div0-and-mod0} \ x \ y & \quad \text{[Function]}
\end{align*}
\]

[R6RS] These are integer division procedures introduced in R6RS. Unlike \text{quotient}, \text{modulo} and \text{remainder}, these procedures can take non-integral values. The dividend \( x \) can be an arbitrary real number, and the divisor \( y \) can be non-zero real number.

\( \text{div} \) returns an integer \( n \), and \( \text{mod} \) returns a real number \( m \), such that:

\[
\begin{itemize}
  \item x = n \cdot y + m, \text{ and}
  \item 0 \leq m < |y|.
\end{itemize}
\]

Examples:

\[
\begin{align*}
(\text{div} \ 123 \ 10) & \Rightarrow 12 \\
(\text{mod} \ 123 \ 10) & \Rightarrow 3 \\
(\text{div} \ 123 \ -10) & \Rightarrow -12 \\
(\text{mod} \ 123 \ -10) & \Rightarrow 3 \\
(\text{div} \ -123 \ 10) & \Rightarrow -13 \\
(\text{mod} \ -123 \ 10) & \Rightarrow 7 \\
(\text{div} \ -123 \ -10) & \Rightarrow 13 \\
(\text{mod} \ -123 \ -10) & \Rightarrow 7 \\
(\text{div} \ 123/7 \ 10/9) & \Rightarrow 15 \\
(\text{mod} \ 123/7 \ 10/9) & \Rightarrow 19/21 \\
& ;; 123/7 = 10/9 \times 15 + 19/21
\end{align*}
\]
(div 14.625 3.75) ⇒ 3.0
(mod 14.625 3.75) ⇒ 3.375
;; 14.625 = 3.75 * 3.0 + 3.375

For a nonnegative integer \( x \) and an integer \( y \), the results of `div` and `mod` matches those of `quotient` and `remainder`. If \( x \) is negative, they differ, though.

`div-and-mod` calculates both `div` and `mod` and returns their results in two values.

`div0` and `mod0` are similar, except the range of \( m \):

- \( x = n \cdot y + m \)
- \( -|y|/2 < m < |y|/2 \)

```
(div0 123 10) ⇒ 12
(mod0 123 10) ⇒ 3
(div0 127 10) ⇒ 13
(mod0 127 10) ⇒ -3
(div0 127 -10) ⇒ -13
(mod0 127 -10) ⇒ -3
(div0 -127 10) ⇒ -13
(mod0 -127 10) ⇒ 3
(div0 -127 -10) ⇒ 13
(mod0 -127 -10) ⇒ 3
```

div0-and-mod0 calculates both `div0` and `mod0` and returns their results in two values.

Here’s a visualization of R6RS and R7RS division and modulo operations: [http://blog.practical-scheme.net/gauche/20100618-integer-divisions](http://blog.practical-scheme.net/gauche/20100618-integer-divisions) It might help to grasp how they works.

```
floor-quotient n d  [Function]
floor-remainder n d [Function]
floor/ n d          [Function]
truncate-quotient n d [Function]
truncate-remainder n d [Function]
truncate/ n d       [Function]
```

[R7RS base] These are integer division operators introduced in R7RS. The names explicitly indicate how they behave when numerator and/or denominator is/are negative.

The arguments \( n \) and \( d \) must be an integer. If any of them are inexact, the result is inexact. If all of them are exact, the result is exact. Also, \( d \) must not be zero.

Given numerator \( n \), denominator \( d \), quotient \( q \) and remainder \( r \), the following relations are always kept.

\[
  r = n - dq \\
  \text{abs}(r) < \text{abs}(d)
\]

Now, `(floor-quotient n d)` and `(truncate-quotient n d)` are the same as `(floor (/ n d))` and `(truncate (/ n d))`, respectively. The `*remainder` counterparts are derived from the above relation.

The `/`-suffixed version, `floor/` and `truncate/`, returns corresponding quotient and remainder as two values.

```
(fLOOR-QUOTIENT 10 -3) ⇒ -4
(fLOOR-REMAINDER 10 -3) ⇒ -2
```
R7RS division library ([scheme.division]) introduces other variation of integer divisions (see Section 10.3.21 [R7RS integer division], page 663).

**gcd** \( n \ldots \) [Function]

**lcm** \( n \ldots \) [Function]

[R7RS base] Returns the greatest common divisor or the least common multiplier of the given integers, respectively.

Arguments must be integers, but doesn’t need to be exact. If any of arguments is inexact, the result is inexact.

**continued-fraction** \( x \) [Function]

Returns a lazy sequence of regular continued fraction expansion of finite real number \( x \). An error is raised if \( x \) is infinite or NaN, or not a real number. The returned sequence is lazy, so the terms are calculated as needed.

\[
\text{(continued-fraction 13579/2468)} \\
\Rightarrow (5 \ 1 \ 1 \ 122 \ 1 \ 9)
\]

\[
(+ \ 5 \ (/ \ (+ \ 1 \ (/ \ (+ \ 1 \ (/ \ (+ \ 122 \ (/ \ (+ \ 1 \ (/ \ 9))))))))))) \\
\Rightarrow 13579/2468
\]

\[
\text{(continued-fraction (exact 3.141592653589793))} \\
\Rightarrow (3 \ 7 \ 15 \ 1 \ 292 \ 1 \ 1 \ 2 \ 1 \ 3 \ 1 \ 14 \ 3 \ 3 \ 2 \ 1 \ 3 \ 7 \ 2 \ 1 \ 1 \ 3 \ 2 \ 42 \ 2)
\]

\[
\text{(continued-fraction 1.5625)} \\
\Rightarrow (1.0 \ 1.0 \ 1.0 \ 3.0 \ 2.0)
\]

**numerator** \( q \) [Function]

**denominator** \( q \) [Function]

[R7RS base] Returns the numerator and denominator of a rational number \( q \).

**rationalize** \( x \) \( ebond \) [Function]

[R7RS base] Returns the simplest rational approximation \( q \) of a real number \( x \), such that the difference between \( x \) and \( q \) is no more than the error bound \( ebond \).

Note that Gauche doesn’t have inexact rational number, so if \( x \) and/or \( ebond \) is inexact, the result is coerced to floating point representation. If you want an exact result, coerce the arguments to exact number first.

\[
\text{(rationalize 1234/5678 1/1000)} \Rightarrow 5/23
\]

\[
\text{(rationalize 3.141592653589793 1/10000)} \\
\Rightarrow 3.141509433962264
\]

\[
\text{(rationalize (exact 3.141592653589793) 1/100000)} \\
\Rightarrow 333/106
\]

\[
\text{(rationalize (exact 3.141592653589793) 1/10000000)} \\
\Rightarrow 75948/24175
\]

;; Some edge cases
\[
\text{(rationalize 2 +inf.0)} \Rightarrow 0
\]
\[
\text{(rationalize +inf.0 0)} \Rightarrow +inf.0
\]
\[
\text{(rationalize +inf.0 +inf.0)} \Rightarrow +nan.0
\]
floor $x$

ceiling $x$

truncate $x$

round $x$

[R7RS base] The argument $x$ must be a real number. Floor and ceiling return a maximum integer that isn’t greater than $x$ and a minimum integer that isn’t less than $x$, respectively. Truncate returns an integer that truncates $x$ towards zero. Round returns an integer that is closest to $x$. If fractional part of $x$ is exactly 0.5, round returns the closest even integer.

Following Scheme’s general rule, the result is inexact if $x$ is an inexact number; e.g. (round 2.3) is 2.0. If you need an exact integer by rounding an inexact number, you have to use exact on the result, or use one of the following procedure ((floor->exact etc).

circlefasf

circlefasf

clamp $x$ :optional min max

Returns

min if $x < min$

$x$ if $min <= x <= max$

max if $max < x$

If min or max is omitted or #f, it is regarded as -inf.0 or +inf.0, respectively. Returns an exact integer only if all the given numbers are exact integers.

(clamp 3.1 0.0 1.0) ⇒ 1.0

(clamp 0.5 0.0 1.0) ⇒ 0.5

(clamp -0.3 0.0 1.0) ⇒ 0.0

(clamp -5 0) ⇒ 0

(clamp 3724 #f 256) ⇒ 256

exp $z$

log $z$

log $z1 z2$

sin $z$

cos $z$

tan $z$

asin $z$

acos $z$

atan $z$

atan y x

[R7RS inexact] Transcendental functions. Work for complex numbers as well. In R7RS, these procedures are in the (scheme inexact) module.

exp returns $e$ raised to the power of $z$. The one-argument version of log returns the natural logarithm of $z$. The two-argument version of log is added in R6RS, and returns base-$z2$ logarithm of $z1$.

The two-argument version of atan returns (angle (make-rectangular x y)) for the real numbers x and y.

See also real-exp etc. below, for real-only version of these functions.

sinh $z$

cosh $z$
Chapter 6: Core library

\[ \text{tanh } z \quad \text{[Function]} \]
\[ \text{asinh } z \quad \text{[Function]} \]
\[ \text{acosh } z \quad \text{[Function]} \]
\[ \text{atanh } z \quad \text{[Function]} \]

Hyperbolic trigonometric functions. Work for complex numbers as well.

See also \texttt{real-sinh} etc. below, for real-only version of these functions.

\[ \text{radians->degrees rad} \quad \text{[Function]} \]
\[ \text{degrees->radians deg} \quad \text{[Function]} \]

Convert radians to degrees and vice versa. The argument must be a real number.

\[ \text{sqrt } z \quad \text{[Function]} \]

[R7RS inexact] Returns a square root of a complex number \( z \). The branch cut scheme is the same as Common Lisp. For real numbers, it returns a positive root.

If \( z \) is the square of an exact real number, the return value is also an exact number.

\[
\begin{align*}
\text{(sqrt 2)} & \Rightarrow 1.4142135623730951 \\
\text{(sqrt -2)} & \Rightarrow 0.0+1.4142135623730951i \\
\text{(sqrt 256)} & \Rightarrow 16 \\
\text{(sqrt 256.0)} & \Rightarrow 16.0 \\
\text{(sqrt 81/169)} & \Rightarrow 9/13
\end{align*}
\]

See also \texttt{real-sqrt} below, for inexact-real-only version of \texttt{sqrt}.

\[ \text{exact-integer-sqrt } k \quad \text{[Function]} \]

[R7RS base] Given an exact nonnegative integer \( k \), returns two exact nonnegative integer \( s \) and \( r \) that satisfy the following equations:
\[
k = (+ (* s s) r) \\
k < (* (+ s 1) (+ s 1))
\]
\[
\text{(exact-integer-sqrt 782763574)} \Rightarrow 27977 \text{ and } 51045
\]

\[ \text{square } z \quad \text{[Function]} \]

[R7RS base] Returns \( (* z z) \).

\[ \text{expt } z1 \ z2 \quad \text{[Function]} \]

[R7RS base] Returns \( z1^{z2} \) (\( z1 \) powered by \( z2 \)), where \( z1 \) and \( z2 \) are complex numbers.

Scheme standard defines \( \text{(expt 0 0)} \) as 1 for convenience.

\[ \text{expt-mod base exponent mod} \quad \text{[Function]} \]

Calculates \( \text{(modulo (expt base exponent) mod)} \) efficiently.

The next example shows the last 10 digits of a mersenne prime \( \text{M}_{74207281} = 2^{74207281} - 1 \)
\[
\text{(- (expt-mod 2 74207281 #e1e10) 1)} \Rightarrow 1086436351
\]

\[ \text{real-exp } x \quad \text{[Function]} \]
\[ \text{real-expt } x \ y \quad \text{[Function]} \]
\[ \text{real-ln } x \quad \text{[Function]} \]
\[ \text{real-sqrt } x \quad \text{[Function]} \]
\[ \text{real-sin } x \quad \text{[Function]} \]
\[ \text{real-cos } x \quad \text{[Function]} \]
\[ \text{real-tan } x \quad \text{[Function]} \]
\[ \text{real-asin } x \quad \text{[Function]} \]
real-acos x  [Function]
real-atan x  [Function]
real-atan y x  [Function]

[SRFI-94] Real-only version of elementary functions. An error is signaled unless the arguments are real numbers.

These are useful for performance-sensitive code, for they don’t need to check whether the input is complex. Furthermore, they are subject to constant folding.

Those names are defined in SRFI-94. However, for the portable programs, you may want to use scheme.flonum module, which is a part of R7RS-large and defines flonum-only versions of elementary functions (see Section 10.3.24 [R7RS flonum], page 670).

Note: SRFI-94 defines real-log, which is a two-argument version of real-only log, but it takes base argument first, which is different from R7RS two-argument log. To avoid confusion, we don’t support real-log in the core.

real-sinpi x  [Function]
real-cospi x  [Function]
real-tanpi x  [Function]

Returns sin(πx), cos(πx), and tan(πx), respectively. An error is signaled if the argument is not a real number. These come handy sometimes, as the following example show:

(use math.const)
(real-sin pi)  ⇒  1.2246467991473532e-16
(real-sinpi 1.0)  ⇒  0.0

(real-cos (/ pi 2))  ⇒  6.123233957367666e-17
(real-cospi 0.5)  ⇒  0.0

real-sinh x  [Function]
real-cosh x  [Function]
real-tanh x  [Function]
real-asinh x  [Function]
real-acosh x  [Function]
real-atanh x  [Function]

Real-only version of hyperbolic functions and their inverses. An error is signaled if the argument is not a real number.

gamma x  [Function]
lgamma x  [Function]

Gamma function and natural logarithmic of absolute value of Gamma function.

NB: Mathematically these functions are defined in complex domain, but currently we only support real number argument.

fixnum-width  [Function]
greatest-fixnum  [Function]
least-fixnum  [Function]

[R6RS] These procedures return the width of fixnum (w), the greatest integer representable by fixnum (2^(w-1) - 1), and the least integer representable by fixnum (~ 2^(w-1)), respectively. You might want to care the fixnum range when you are writing a performance-critical section.

These names are defined in R6RS. Common Lisp and ChezScheme have most-positive-fixnum and most-negative-fixnum.

NB: Before 0.9.5, fixnum-width had a bug to return one smaller than the supposed value.
6.3.5 Numerical conversions

**make-rectangular** $x_1 \ x_2$  \[ \text{Function} \]

**make-polar** $x_1 \ x_2$  \[ \text{Function} \]

[R7RS complex] Creates a complex number from two real numbers, $x_1$ and $x_2$. **make-rectangular** returns $x_1 + i x_2$. **make-polar** returns $x_1 e^{i x_2}$.

In R7RS, these procedures are in the (scheme complex) library.

**real-part** $z$  \[ \text{Function} \]

**imag-part** $z$  \[ \text{Function} \]

**magnitude** $z$  \[ \text{Function} \]

**angle** $z$  \[ \text{Function} \]

[R7RS complex] Decompose a complex number $z$ and returns a real number. **real-part** and **imag-part** return $z$’s real and imaginary part, respectively. **magnitude** and **angle** return $z$’s magnitude and angle, respectively.

In R7RS, these procedures are in the (scheme complex) library.

**decode-float** $x$  \[ \text{Function} \]

For a given finite floating-point number, returns a vector of three exact integers, $(m, e, \text{sign})$, where

$$ x = (* \text{sign} m (\text{expt} 2.0 e)) $$

\text{sign} is either 1, 0 or -1.

If $x$ is $+inf.0$ or $-inf.0$, $m$ is #t. If $x$ is $+nan.0$, $m$ is #f.

The API is taken from ChezScheme.

```scheme
(decode-float 3.1415926)
⇒ #(7074237631354954 -51 1)
(* 7074237631354954 (expt 2.0 -51))
⇒ 3.1415926
```

```scheme
(decode-float +nan.0)
⇒ #(#f 0 -1)
```

**encode-float** vector  \[ \text{Function} \]

This is an inverse of **decode-float**. Vector must be a three-element vector as returned from **decode-float**.

```scheme
(encode-float #\((7074237631354954 -51 1)\))
⇒ 3.1415926
```

```scheme
(encode-float #\(\#t 0 1)\))
⇒ +inf.0
```

**fmod** $x \ y$  \[ \text{Function} \]

**modf** $x$  \[ \text{Function} \]

**frexp** $x$  \[ \text{Function} \]

**ldexp** $x \ n$  \[ \text{Function} \]

[POSIX] These procedures can be used to compose and decompose floating point numbers. **Fmod** computes the remainder of dividing $x$ by $y$, that is, it returns $x - n^*y$ where $n$ is the quotient of $x/y$ rounded towards zero to an integer. **Modf** returns two values; a fractional part of $x$ and an integral part of $x$. **Frexp** returns two values, **fraction** and **exponent** of $x$, where $x = \text{fraction} * 2^{\text{exponent}}$, and $0.5 <= |\text{fraction}| < 1.0$, unless $x$ is zero. (When $x$ is
zero, both \textit{fraction} and \textit{exponent} are zero). \texttt{Ldexp} is a reverse operation of \texttt{frexp}; it returns a real number \(x \times 2^n\).

\[
\text{(fmod 32.1 10.0) } \Rightarrow 2.1 \\
\text{(fmod 1.5 1.4) } \Rightarrow 0.1 \\
\text{(modf 12.5) } \Rightarrow 0.5 \text{ and } 12.0 \\
\text{(frexp 3.14) } \Rightarrow 0.785 \text{ and } 2 \\
\text{(ldexp 0.785 2) } \Rightarrow 3.14
\]

\textbf{exact} \(z\) \hspace{1cm} \textbf{inexact} \(z\) \hspace{1cm} \textbf{Function}

[R7RS base] Returns an exact or an inexact representation of the given number \(z\), respectively. Passing an exact number to \texttt{exact}, and an inexact number to \texttt{inexact}, are no-op.

Gauche doesn’t have exact complex number with non-zero imaginary part, nor exact infinites and NaNs, so passing those to \texttt{exact} raises an error.

\[
\text{(inexact 1) } \Rightarrow 1.0 \\
\text{(inexact 1/10) } \Rightarrow 0.1
\]

If an inexact finite real number is passed to \texttt{exact}, the simplest exact rational number within the precision of the floating point representation is returned.

\[
\text{(exact 1.0) } \Rightarrow 1 \\
\text{(exact 0.1) } \Rightarrow 1/10 \\
\text{(exact (/ 3.0)) } \Rightarrow 1/3
\]

For all finite inexact real number \(x\), \((\text{inexact (exact } x))\) is always \texttt{eqv?} to the original number \(x\).

(Note that the inverse doesn’t hold, that is, an exact number \(n\) and \((\text{exact (inexact } n))\) aren’t necessarily the same. It’s because many (actually, infinite number of) exact numbers can be mapped to one inexact number.)

To specify the error tolerance when converting inexact real numbers to exact rational numbers, use \texttt{rationalize} or \texttt{real->rational}.

\textbf{exact->inexact} \(z\) \hspace{1cm} \textbf{inexact->exact} \(z\) \hspace{1cm} \textbf{Function}

[R5RS] Converts exact number to inexact one, and vice versa.

In fact, \texttt{exact->inexact} returns the argument as is if an inexact number is passed, and \texttt{inexact->exact} returns the argument if an exact number is passed, so in Gauche they are equivalent to \texttt{inexact} and \texttt{exact}, respectively. Note that other R5RS implementation may raise an error if passing an inexact number to \texttt{exact->inexact}, for example.

Generally \texttt{exact} and \texttt{inexact} are preferred, for they are more concise, and you don’t need to care whether the argument is exact or inexact numbers. These procedures are for compatibility with R5RS programs.

\textbf{real->rational} \(x :\text{optional hi lo open?}\) \hspace{1cm} \textbf{Function}

Find the simplest rational representation of a finite real number \(x\) within the specified error bounds. This is the low-level routine called by \texttt{rationalize} and \texttt{exact}. Typically you want to use \texttt{rationalize} (see Section 6.3.4 [Arithmetics], page 126) for this purpose. Use \texttt{real->rational} only when you need finer control of error bounds.

The result rational value \(r\) satisfies the following condition:

\[
(\leq (- x \text{ lo}) r (+ x \text{ hi})) \quad \text{when open? is #f} \\
(< (- x \text{ lo}) r (+ x \text{ hi})) \quad \text{otherwise}
\]

Note that both \texttt{hi} and \texttt{lo} must be nonnegative.
If hi and/or lo is omitted, it is determined by x: if x is exact, hi and lo are defaulted to zero; if x is inexact, hi and lo depend on the precision of the floating point representation of x. In the latter case, the open? also depends on x—it is true if the mantissa of x is odd, and false otherwise, reflecting the round-to-even rule. So, if you call real->rational with one finite number, you’ll get the same result as exact:

(real->rational 0.1) ⇒ 1/10

Passing zeros to the error bounds makes it return the exact conversion of the floating number itself (that is, the exact calculation of (* sign mantissa (expt 2 exponent))).

(real->rational 0.1 0 0) ⇒ 3602879701896397/36028797018963968

(If you give both hi and lo, but omit open?, we assume closed range.)

number->string z :optional radix use-upper?  
string->number string :optional radix default-exactness  
[Function]  
[R7RS+] These procedures convert a number and its string representation in radix radix system. radix must be between 2 and 36 inclusive. If radix is omitted, 10 is assumed.

Number->string takes a number z and returns a string. If z is not an exact integer, radix must be 10. For the numbers with radix more than 10, lower case alphabet character is used for digits, unless the optional argument use-upper? is true, in that case upper case characters are used. The argument use-upper? is Gauche’s extension.

String->number takes a string string and parses it as a number in radix radix system. If the number contains a decimal point, only radix 10 is allowed. If the given string can’t be a number, #f is returned.

The default-exactness optional argument of string->number is Gauche’s extension, and it must be either #f (default), a symbol exact, or a symbol inexact. If it is either symbol, it sets the exactness of the number if no exactness prefix (#e or #i) is given. If string has an exactness prefix, default-exactness is irrelevant.

(string->number "2.718281828459045" 10 'exact)  ⇒ 543656365691809/200000000000000
(string->number "#i2.718281828459045" 10 'exact)  ⇒ 2.718281828459045
(string->number "1/3" 10 'inexact)  ⇒ 0.3333333333333333
(string->number "#e1/3" 10 'inexact)  ⇒ 1/3

x->number obj  
[Generic Function]  
x->integer obj  
[Generic Function]  
Generic coercion functions. Returns ‘natural’ interpretation of obj as a number or an exact integer, respectively. The default methods are defined for numbers and strings; a string is interpreted by string->number, and if the string can’t be interpreted as a number, 0 is returned. Other obj is simply converted to 0. If obj is naturally interpreted as a number that is not an exact integer, x->integer uses round and inexact->exact to obtain an integer.

Other class may provide a method to customize the behavior.

6.3.6 Basic bitwise operations  
These procedures treat integers as half-open bit vectors. If an integer is positive, it is regarded as if infinite number of zeros are padded to the left. If an integer is negative, it is regarded in 2’s complement form, and infinite number of 1’s are padded to the left.

In regard to the names of those operations, there are two groups in the Scheme world; Gauche follows the names of the original SLIB’s “logical” module, which was rooted in CL. Another group uses a bit long but descriptive name such as arithmetic-shift.
R7RS bitwise library (see Section 10.3.22 [R7RS bitwise operations], page 664) provides additional bitwise operations.

\textbf{ash \, n \, count}  
\textit{[Function]}  
\text{[SRFI-60]} Shifts integer \, n \, left with \, count \, bits. If \, count \, is negative, \text{ash} \, shifts \, n \, right with \, -\text{count} \, bits.  

\begin{verbatim}
; Note: 6 \equiv \ldots00110, and  
; -6 \equiv \ldots11010  
(ash 6 2) \Rightarrow 24 \; ;[...0011000]  
(ash 6 -2) \Rightarrow 1 \; ;[...0000001]  
(ash -6 2) \Rightarrow -24 \; ;[...1101000]  
(ash -6 -2) \Rightarrow -2 \; ;[...1111110]
\end{verbatim}

\textbf{logand \, n1 \ldots}  
\textit{[Function]}  
\textbf{logior \, n1 \ldots}  
\textit{[Function]}  
\textbf{logxor \, n1 \ldots}  
\textit{[Function]}  
\text{[SRFI-60]} Returns bitwise and, bitwise inclusive or and bitwise exclusive or of integers \, n1 \ldots. If no arguments are given, \text{logand} \, returns -1, and \text{logior} \, and \text{logxor} \, returns 0.

\textbf{lognot \, n}  
\textit{[Function]}  
\text{[SRFI-60]} Returns bitwise not of an integer \, n.

\textbf{logtest \, n1 \, n2 \ldots}  
\textit{[Function]}  
\text{[SRFI-60]} \equiv (\text{not} \, (\text{zero?} \, (\text{logand} \, n1 \, n2 \ldots)))

\textbf{logbit? \, index \, n}  
\textit{[Function]}  
\text{[SRFI-60]} Returns \#t \, if \, index-th bit of integer \, n \, is 1, \#f otherwise.

\textbf{logset+clear \, n \, set-bits \, clear-bits}  
\textit{[Function]}  
Returns an integer such that \, n's \, bit is set where \, set-bits \, is 1, and cleared where \, clear-bits \, is 1. If a bit is included in both \, set-bits \, and \, clear-bits, \, it is cleared.  
It is effectively the same as \, (logand \, (logior \, n \, set-bits) \, (lognot \, clear-bits)).  
\begin{verbatim}
(logset+clear \#b11000011 \#b00001100 \#b01000001)  
\Rightarrow 142 \; ;; \#b10001110
\end{verbatim}

\textbf{bit-field \, n \, start \, end}  
\textit{[Function]}  
\textbf{[R7RS bitwise]} Extracts \, start-th \, bit (inclusive) to \, end-th \, bit (exclusive) from an exact integer \, n, where \, start < \, end.

\textbf{copy-bit \, index \, n \, bit}  
\textit{[Function]}  
\textbf{[R7RS bitwise]} If \, bit \, is true, sets \, index-th \, bit of an exact integer \, n. If \, bit \, is false, resets \, index-th \, bit of an exact integer \, n.

\textbf{copy-bit-field \, n \, from \, start \, end}  
\textit{[Function]}  
\textbf{[SRFI-60]} Returns an exact integer, each bit of which is the same as \, n \, except the \, start-th \, bit (inclusive) to \, end-th \, bit (exclusive), which is a copy of the lower \, (end-start) \,-th \, bits of an exact integer \, from.  
\begin{verbatim}
(number->string (copy-bit-field \#b10000000 -1 1 5) 2)  
\Rightarrow "10011110"
\end{verbatim}
\begin{verbatim}
(number->string (copy-bit-field \#b10000000 \#b010101010 1 7) 2)  
\Rightarrow "11010100"
\end{verbatim}

Note: The API of this procedure was originally taken from SLIB, and at that time, the argument order was \, (copy-bit-field \, n \, start \, end \, from). During the discussion of SRFI-60
the argument order was changed for the consistency, and the new versions of SLIB followed it. We didn’t realize the change until recently - before 0.9.4, this procedure had the old argument order. Code that is using this procedure needs to be fixed. If you need your code to work with both versions of Gauche, have the following definition in your code.

\[
\begin{align*}
\text{(define (copy-bit-field to from start end)} & \\
& \quad (\text{if (< start end)} & \\
& & \quad \quad \begin{align*}
& \quad \quad \begin{align*}
& \quad \quad \quad \text{(let1 mask (- (ash 1 (- end start)) 1)} & \\
& \quad \quad \quad \quad \text{(logior (logand to (lognot (ash mask start)))} & \\
& \quad \quad \quad \quad \quad \text{(ash (logand from mask) start))} & \\
& \quad \quad \quad \quad \quad \text{from))} & \\
\end{align*}
& \quad \quad \end{align*}
& \quad \end{align*}
\end{align*}
\]

\textbf{logcount \( n \)}

[SRFI-60] If \( n \) is positive, returns the number of 1’s in the bits of \( n \). If \( n \) is negative, returns the number of 0’s in the bits of 2’s complement representation of \( n \).

\[
\begin{align*}
(\text{logcount 0}) & \Rightarrow 0 \\
(\text{logcount #b0010}) & \Rightarrow 1 \\
(\text{logcount #b0110}) & \Rightarrow 2 \\
(\text{logcount #b1111}) & \Rightarrow 4 \\
(\text{logcount #b-0001}) & \Rightarrow 0 \text{ ;; 2’s complement: \ldots111111} \\
(\text{logcount #b-0010}) & \Rightarrow 1 \text{ ;; 2’s complement: \ldots111110} \\
(\text{logcount #b-0011}) & \Rightarrow 1 \text{ ;; 2’s complement: \ldots111101} \\
(\text{logcount #b-0100}) & \Rightarrow 2 \text{ ;; 2’s complement: \ldots111100}
\end{align*}
\]

\textbf{integer-length \( n \)}

[R7RS bitwise] Returns the minimum number of bits required to represent an exact integer \( n \). Negative integer is assumed to be in 2’s complement form. A sign bit is not considered.

\[
\begin{align*}
(\text{integer-length 255}) & \Rightarrow 8 \\
(\text{integer-length 256}) & \Rightarrow 9 \\
(\text{integer-length -256}) & \Rightarrow 8 \\
(\text{integer-length -257}) & \Rightarrow 9
\end{align*}
\]

\textbf{twos-exponent \( n \)}

If \( n \) is a power of two, that is, \((\text{expt } 2 \ k)\) and \( k >= 0 \), then returns \( k \). Returns \#f if \( n \) is not a power of two.

\textbf{twos-exponent-factor \( n \)}

Returns maximum \( k \) such that \((\text{expt } 2 \ k)\) is a factor of \( n \). In other words, returns the number of consecutive zero bits from LSB of \( n \). When \( n \) is zero, we return \(-1\) for the consistency of the following equivalent expression.

This can be calculated by the following expression; this procedure is for speed to save creating intermediate numbers when \( n \) is bignum.

\[
(- \text{(integer-length \{logxor n (- n 1)\))) 1)}
\]

This procedure is also equivalent to SRFI-60’s \texttt{log2-binary-factors} and \texttt{first-set-bit} (see Section 11.14 [Integers as bits], page 718).

\textbf{6.3.7 Endianness}

In the Scheme world you rarely need to know about how the numbers are represented inside the machine. However, it matters when you have to exchange data to/from the outer world in binary representation.
Gauche’s binary I/O procedures, such as in the binary.io module (see Section 12.1 [Binary I/O], page 808) and write-uvector/read-uvector! (see Section 6.13.2 [Uniform vectors], page 198), take optional endian argument to specify the endianness.

Currently Gauche recognizes the following endiannesses.

**big-endian**

**big** Big-endian. With this endianness, a 32-bit integer \texttt{#x12345678} will be written out as an octet sequence \texttt{#x12 #x34 #x56 #x78}.

Gauche has been using big-endian, but scheme.bytevector incorporated in R7RS uses big, so we recognize both.

**little-endian**

**little** Little-endian. With this endianness, a 32-bit integer \texttt{#x12345678} is written out as an octet sequence \texttt{#x78 #x56 #x34 #x12}.

Gauche has been using little-endian, but scheme.bytevector incorporated in R7RS uses little, so we recognize both.

**arm-little-endian**

This is a variation of little-endian, and used in ARM processors in some specific modes. It works just like little-endian, except reading/writing double-precision floating point number (f64), which is written as two little-endian 32bit words ordered by big-endian (e.g. If machine register’s representation is \texttt{#x0102030405060708}, it is written as \texttt{#x04 #x03 #x02 #x01 #x08 #x07 #x06 #x05}.

When the endian argument is omitted, those procedures use the parameter default-endian:

**default-endian**

This is a dynamic parameter (see Section 6.16.1 [Parameters], page 233) to specify the endianness the binary I/O routines use when its endian argument is omitted. The initial value of this parameter is the system’s native endianness.

The system’s native endianness can be queried with the following procedure:

**native-endian**

Returns a symbol representing the system’s endianness.

### 6.4 Booleans

**<boolean>**

A boolean class. Only \#t and \#f belong to this class.

**not obj**

[R7RS base] Returns \#t if and only if obj is \#f, and returns \#f otherwise.

**boolean? obj**

[R7RS base] Returns \#t if obj is a boolean value.

**boolean obj**

[SRFI-235] Returns \#f iff obj is \#f, and returns \#t otherwise. Convenient to coerce a value to boolean.

**boolean=? a b c ...**

[R7RS base] Every argument must be a boolean value. Returns \#t iff all values are the same, \#f otherwise.
6.5 Undefined values

While working with Gauche, sometimes you encounter a value printed as #<undef>, an undefined value.

    gosh> (if #f #t) #<undef>

It is a value used as a filler where the actual value doesn’t matter, or there’s no other suitable value, or the binding hasn’t been calculated.

Do not confuse undefined values with unbound variables: A variable can be bound to #<undef>, for it is just an ordinary first-class value. On the other hand, an unbound variable means there’s no value associated with the variable.

However, #<undef> may be used in certain occasions to indicate that a value is not provided for the variable. For example, the toplevel variable can be bound to #<undef> if it is defined by (define variable) form (see Section 4.10 [Definitions], page 68). An optional procedure parameter without default value is bound to #<undef> if an actual argument is not given (see Section 4.3 [Making procedures], page 47).

Note that it cannot be distinguished from the case a value is actually provided, and the value just happens to be #<undef>. If you get an #<undef>, you can say at most is that the value doesn’t matter. You shouldn’t let it carry too much meanings.

The #<undef> value is counted as true value in generalized boolean context, since it is not #f. However, branching based on #<undef> is dangerous—a procedure that is defined to return unspecified value may merely returning #<undef> as a provisional value; it will change the return value in future. Since the return value isn’t specified, no one should be using it. The code that tests such result value as a generalized boolean may break if the procedure changes the return value.

In fact, we’ve found that there are quite a few code that accidentally tests #<undef> return value in conditionals. They can be seeds for future bugs, so we added a feature to warn when #<undef> value is used in the test of branches. You can turn it on with setting the environment variable GAUCHE_CHECK_UNDEFINED_TEST. In future, we may turn it on while testing.

One typical case of such accidental use of #<undef> branching is in and-let*; the following code assumes print always return #<undef>, which is counted as a true value, and expects the control to proceed to the next clause. It’ll break if print ever changes so that it may return #f in some cases.

    (and-let* ([var (foo x y z)]
               [ (print var) ] ;; branch on #<undef>
               [baz (bar var)])
      ...

Being said that, there are a couple of procedures to deal with undefined values.

**undefined?** obj  
Returns #t iff obj is an undefined value.

**undefined**  
Returns an undefined value.

6.6 Pairs and lists

Pairs and lists are one of the most fundamental data structure in Scheme. Gauche core provides all standard list procedures, plus some useful procedures that are commonly supported in lots of implementations. If they are not enough, you can find more procedures in the modules described in Section 10.3.1 [R7RS lists], page 593, and Section 12.81 [Combination library], page 1015.
See also Section 9.6 [Collection framework], page 404, and Section 9.31 [Sequence framework], page 510, for generic collection/sequence operations.

6.6.1 Pair and null class

<list> [Builtin Class]
An abstract class represents lists. A parent class of <null> and <pair>. Inherits <sequence>.
Note that a circular list is also an instance of the <list> class, while list? returns false on the circular lists and dotted lists.

(use srfi.1)
(list? (circular-list 1 2)) ⇒ #f
(is-a? (circular-list 1 2) <list>) ⇒ #t

<null> [Builtin Class]
A class of empty list. () is the only instance.

<pair> [Builtin Class]
A class of pairs.

6.6.2 Mutable and immutable pairs
A pair may be either mutable or immutable. You can desctructively modify mutable pairs with set-car!, set-cdr!, or other destructive procedures (usually they have ! at the end). An error is signaled when you try to modify an immutable pair.

In Gauche, both type of pairs can be treated the same unless you try to modify them. Both satisfies the predicate pair?. If you need to test specifically if a pair is immutable, use ipair? (see Section 6.6.3 [List predicates], page 141). The traditional constructor cons creates a mutable pair; you can create an immutable pair with ipair (see Section 6.6.4 [List constructors], page 142). More procedures that use immutable pairs are defined in R7RS-large scheme.ilist module (see Section 10.3.8 [R7RS immutable lists], page 621).

Note that quoted literals are immutable. Old versions of Gauche wasn’t supported immutable pairs and quoted literal lists were mutable. If your code accidentally mutate such pairs, you’ll get an error. If you need to run such code without tracking down the error, you can define the environment variable GAUCHE_MUTABLE_LITERALS.

6.6.3 List predicates

pair? obj [Function]
[R7RS base] Returns #t if obj is a pair, #f otherwise.

ipair? obj [Function]
[R7RS ilist] Returns #t iff obj is an immutable pair, #f otherwise.

An immutable pair is indistinguishable from a mutable pair except using this predicate, or when you attempt to modify it. See Section 6.6.2 [Mutable and immutable pairs], page 141.

null? obj [Function]
[R7RS base] Returns #t if obj is an empty list, #f otherwise.

null-list? obj [Function]
[R7RS list] Returns #t if obj is an empty list, #f if obj is a pair. If obj is neither a pair nor an empty list, an error is signaled.

This can be used instead of null? to check the end-of-list condition when you want to be more picky about non-proper lists.
list? obj
[Function]
[R7RS base] Returns #t if obj is a proper list, #f otherwise. This function returns #f if obj is a dotted or circular list.
See also proper-list?, circular-list? and dotted-list? below.

proper-list? x
[Function]
[R7RS list] Returns #t if x is a proper list, that is, a finite list terminated by ().

circular-list? x
[Function]
[R7RS list] Returns #t if x is a circular list. A list is circular if you follow cdr of the pairs you’ll eventually get to a pair you already visited. It doesn’t necessary that the head of the list x is a part of the circle. A list isn’t circular by the cycle that involves car of the pairs.

dotted-list? x
[Function]
[R7RS list] Returns #t if x is a finite, non-nil-terminated list. This includes non-pair, non-() values (e.g. symbols, numbers), which are considered to be dotted lists of length 0.

6.6.4 List constructors

cons obj1 obj2
[Function]
[R7RS base] Constructs a mutable pair of obj1 and obj2 and returns it.
(cons 'a 'b) ⇒ (a . b)

ipair obj1 obj2
[Function]
[R7RS ilist] Constructs an immutable pair of obj1 and obj2 and returns it.
(ipair 'a 'b) ⇒ (a . b)

make-list len :optional fill
[Function]
[R7RS base] Makes a proper list of length len. If optional argument fill is provided, each element is initialized by it. Otherwise each element is undefined.
(make-list 5 #t) ⇒ (#t #t #t #t #t)

list obj ... 
[Function]
[R7RS base] Makes a list, whose elements are obj ....
(list 1 2 3) ⇒ (1 2 3)
(list) ⇒ ()

ilist obj ... 
[Function]
[R7RS ilist] Makes a list, whose elements are obj ..., and which consists of immutable pairs.
(ilist 1 2 3) ⇒ (1 2 3)
(ilist) ⇒ ()

(list-set! (ilist 1 2 3) 1 'a)
⇒ ERROR: Attempt to modify an immutable pair: (2 3)

list* obj1 obj2 ...
[Function]
cons* obj1 obj2 ...
[Function]
[R7RS list] Like list, but the last argument becomes cdr of the last pair. Two procedures are exactly the same. Gauche originally had list*, and SRFI-1 (R7RS (scheme list)) defines cons*.
(list* 1 2 3) ⇒ (1 2 . 3)
(list* 1) ⇒ 1
list-copy list  
[Function]  
[R7RS base] Shallow copies list. If list is circular, an error is thrown. (Detecting circular list is Gauche’s extension; R7RS allows the procedure to diverge.)

iota count :optional (start 0) (step 1)  
[Function]  
[R7RS list] Returns a list of count numbers, starting from start, increasing by step. Count must be a nonnegative integer. If both start and step are exact, the result is a list of exact numbers; otherwise, it is a list of inexact numbers.

(iota 5) ⇒ (0 1 2 3 4)  
(iota 5 1 3/7) ⇒ (1 10/7 13/7 16/7 19/7)  
(iota 5 0 -0.1) ⇒ (0 -0.1 -0.2 -0.3 -0.4)  

This creates a list eagerly. If the list is short it is fast enough, but if you want to count tens of thousands of numbers, you may want to do so lazily. See iota (see Section 6.18.2 [Lazy sequences], page 241).

cond-list clause . . .  
[Macro]  
Construct a list by conditionally adding entries. Each clause has a test and expressions. When its test yields true, the result of associated expression is used to construct the resulting list. When the test yields false, nothing is inserted.

Clause must be either one of the following form:

(test expr ...)  
Test is evaluated, and when it is true, expr . . . are evaluated, and the return value becomes a part of the result. If no expr is given, the result of test is used if it is not false.

(test => proc)  
Test is evaluated, and when it is true, proc is called with the value, and the return value is used to construct the result.

(test @ expr ...)  
Like (test expr ...), except that the result of the last expr must be a list, and it is spliced into the resulting list, like unquote-splicing.

(test => @ proc)  
Like (test => proc), except that the result of proc must be a list, and and it is spliced into the resulting list, like unquote-splicing.

(let ((alist '((x 3) (y -1) (z 6))))  
(cond-list ((assoc 'x alist) 'have-x)  
  ((assoc 'w alist) 'have-w)  
  ((assoc 'z alist) => cadr)))  
⇒ (have-x 6)  

(let ((x 2) (y #f) (z 5))  
  (cond-list (x @ '(:x ,x))  
    (y @ '(:y ,y))  
    (z @ '(:z ,z))))  
⇒ (:x 2 :z 5)  

6.6.5 List accessors and modifiers

car pair  
[Function]  
[R7RS base] Returns car of pair, respectively.

cdr pair  
[Function]  
[R7RS base] Returns cdr of pair, respectively.
Chapter 6: Core library

**set-car!** `pair obj`  
[Function]

**set-cdr!** `pair obj`  
[Function]

[R7RS base] Modifies car and cdr of `pair`, by `obj`, respectively.

Note: `(setter car) ≡ set-car!`, and `(setter cdr) ≡ set-cdr!`.

**caar** `pair`  
[Function]

**cadr** `pair`  
...  

**cdddar** `pair`  
[Function]

**cddddr** `pair`  
[Function]

[R7RS base][R7RS cxr] `caar` ≡ `(car (car x))`, `cadr` ≡ `(car (cdr x))`, and so on.

In R7RS, more than two-level of accessors are defined in the `(scheme cxr)` library.

The corresponding setters are also defined.

```scheme
(let ((x (list 1 2 3 4 5)))
  (set! (caddr x) -1)
  x)
⇒ (1 2 -1 4 5)
```

**length** `list`  
[Function]

[R7RS base] Returns the length of a proper list `list`. If `list` is a dotted list, an error is signaled.

If `list` is a circular list, this function diverges.

**length+** `x`  
[Function]

[R7RS list] If `x` is a proper list, returns its length. For all other `x`, including a circular list, it returns `#f`.

If you want a length of dotted list in terms of the number of pairs, use `num-pairs` below.

**length=?** `x k`  
[Function]

**length<=?** `x k`  
[Function]

**length<=?** `x k`  
[Function]

**length>?** `x k`  
[Function]

**length=>?** `x k`  
[Function]

Returns `#t` iff `x` is a (possibly improper) list whose length is equal to, less than, less than or equal to, greater than, or greater than or equal to an exact integer `k`, respectively. This procedure only follows the list up to the `k` items, so it doesn’t realize elements of lazy sequence more than needed (See Section 6.18.2 [Lazy sequences], page 241, for the lazy sequences).

Dotted lists and circular lists are allowed. For the dotted list, the cdr of the last pair isn’t counted; that is, a non-pair object has length 0, and `(a . b)` has length 1. A circular list is treated as if it has infinite length.

```scheme
(length<=? '(a b) 2) ⇒ #t
(length<=? '(a b) 1) ⇒ #f
(length<=? '() 0) ⇒ #t
```

### ddd airports

```scheme
(let ((x (list 1 2 3 4 5)))
  (set! (caddr x) -1)
  x)
⇒ (1 2 -1 4 5)
```

**length** `list`  
[Function]

[R7RS base] Returns the length of a proper list `list`. If `list` is a dotted list, an error is signaled.

If `list` is a circular list, this function diverges.

**length+** `x`  
[Function]

[R7RS list] If `x` is a proper list, returns its length. For all other `x`, including a circular list, it returns `#f`.

If you want a length of dotted list in terms of the number of pairs, use `num-pairs` below.

**length=?** `x k`  
[Function]

**length<=?** `x k`  
[Function]

**length<=?** `x k`  
[Function]

**length>?** `x k`  
[Function]

**length=>?** `x k`  
[Function]

Returns `#t` iff `x` is a (possibly improper) list whose length is equal to, less than, less than or equal to, greater than, or greater than or equal to an exact integer `k`, respectively. This procedure only follows the list up to the `k` items, so it doesn’t realize elements of lazy sequence more than needed (See Section 6.18.2 [Lazy sequences], page 241, for the lazy sequences).

Dotted lists and circular lists are allowed. For the dotted list, the cdr of the last pair isn’t counted; that is, a non-pair object has length 0, and `(a . b)` has length 1. A circular list is treated as if it has infinite length.

```scheme
(length<=? '(a b) 2) ⇒ #t
(length<=? '(a b) 1) ⇒ #f
(length<=? '() 0) ⇒ #t
```

;; ddd airports

```scheme
(let ((x (list 1 2 3 4 5)))
  (set! (caddr x) -1)
  x)
⇒ (1 2 -1 4 5)
```

NB: The name of these procedures might be misleading, for other procedures with the name `something<=>?` etc. usually takes objects of the same type. We don’t have any better idea now, unfortunately.
**num-pairs** *lis*

Returns the number of pairs that consists a (proper, dotted, or circular) list *lis*, in an exact integer. If *lis* is a proper list, it is the same as `length`.

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(num-pairs '(a b c d e))</code></td>
<td>5</td>
</tr>
<tr>
<td><code>(num-pairs '())</code></td>
<td>0</td>
</tr>
<tr>
<td><code>(num-pairs '(a b c d . e))</code></td>
<td>4</td>
</tr>
<tr>
<td><code>(num-pairs 'a)</code></td>
<td>0</td>
</tr>
<tr>
<td><code>(num-pairs '#0=(a b . #0#))</code></td>
<td>2</td>
</tr>
<tr>
<td><code>(num-pairs '(a . #0=(b c . #0#)))</code></td>
<td>3</td>
</tr>
</tbody>
</table>

**take** *x i*  
**drop** *x i*

[R7RS list] **take** returns the first *i* elements of list *x*. **drop** returns all but the first *i* elements of list *x*.

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(take '(a b c d e) 2)</code></td>
<td>(a b)</td>
</tr>
<tr>
<td><code>(take '(a b c d e) 2)</code></td>
<td>(c d e)</td>
</tr>
</tbody>
</table>

*x* may be any value:

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(take '(1 2 3 . d) 2)</code></td>
<td>(1 2)</td>
</tr>
<tr>
<td><code>(drop '(1 2 3 . d) 2)</code></td>
<td>(3 . d)</td>
</tr>
<tr>
<td><code>(drop '(1 2 3 . d) 3)</code></td>
<td>d</td>
</tr>
</tbody>
</table>

**drop** is exactly equivalent to performing *i* cdr operations on *x*. The returned value shares a common tail with *x*. On the other hand, take always allocates a new list for result if the argument is a list of non-zero length.

An error is signaled if *i* is past the end of list *x*. See **take** and **drop** below for more tolerant version.

For generic subsequence extraction from any sequence, see **subseq** in Section 9.31.2 [Slicing sequence], page 511.

**take** *list k* :optional (fill? #f) (padding #f)  
**drop** *list k*

More tolerant version of **take** and **drop**. They won’t raise an error even if *k* is larger than the size of the given list.

If the list is shorter than *k* elements, **take** returns a copy of *list* by default. If *fill?* is true, *padding* is added to the result to make its length *k*.

On the other hand, **drop** just returns an empty list when the input list is shorter than *k* elements.

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(take* '(a b c d) 3)</code></td>
<td>(a b c)</td>
</tr>
<tr>
<td><code>(take* '(a b c d) 6)</code></td>
<td>(a b c d)</td>
</tr>
<tr>
<td><code>(take* '(a b c d) 6 #t)</code></td>
<td>(a b c d #f #f)</td>
</tr>
<tr>
<td><code>(take* '(a b c d) 6 #t 'z)</code></td>
<td>(a b c d z z)</td>
</tr>
<tr>
<td><code>(drop* '(a b c d) 3)</code></td>
<td>(d)</td>
</tr>
<tr>
<td><code>(drop* '(a b c d) 5)</code></td>
<td>()</td>
</tr>
</tbody>
</table>

Note: For generic subsequence extraction from any sequence, see **subseq** in Section 9.31.2 [Slicing sequence], page 511.

**take-right** *lis k*  
**drop-right** *lis k*

[R7RS list] **take-right** returns the last *k* elements of *lis*. **drop-right** returns all but the last *k* elements of *lis*.

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(take-right '(a b c d e) 2)</code></td>
<td>(d e)</td>
</tr>
</tbody>
</table>
(drop-right '(a b c d e) 2) => (a b c)

lis may be any finite list.

(take-right '(1 2 3 . d) 2) => (2 3 . d)
(drop-right '(1 2 3 . d) 2) => (1)
(take-right '(1 2 3 . d) 0) => d
(drop-right '(1 2 3 . d) 0) => (1 2 3)

take-right's return value always shares a common tail with lis. drop-right always allocates a new list if the argument is a list of non-zero length.

An error is signaled if k is larger than the length of lis. See take-right* and drop-right* below, for more tolerant version.

take-right* list k :optional (fill? #f) (padding #f) [Function]
drop-right* list k [Function]

Like take* and drop*, but counts from right of list. If list is shorter than k elements, they won’t raise an error. Instead, drop-right* just returns an empty list, and take-right* returns list itself by default. If fill? is true for take-right*, padding is added on the left of the result to make its length k. The result still shares the list.

take! lis k [Function]
drop-right! lis k [Function]

[R7RS list] Linear update variants of take and drop-right. Those procedures may destructively modifies lis.

If lis is circular, take! may return a list shorter than expected.

list-tail list k :optional fallback [Function]

[R7RS base] Returns k-th cdr of list. list can be a proper, dotted or circular list. (If list is a dotted list, its last cdr is simply ignored).

If k is negative or larger than the length of list, the behavior depends on whether the optional fallback argument is given or not. If fallback is given, it is returned. Otherwise, an error is signaled.

list-ref list k :optional fallback [Function]

[R7RS +] Returns k-th element of list. list can be a proper, dotted or circular list.

By default, list-ref signals an error if k is negative, or greater than or equal to the length of list. However, if an optional argument fallback is given, it is returned for such case. This is an extension of Gauche.

list-set! list k v [Function]

[R7RS base] Modifies the k-th element of a list by v. It is an error unless k is an exact integer between 0 and one minus the length of k. If list is immutable, no error is signalled but the behavior is undefined.

last-pair list [Function]

[R7RS list] Returns the last pair of list. list can be a proper or dotted list.

(last-pair '(1 2 3)) => (3)
(last-pair '(1 2 . 3)) => (2 . 3)
(last-pair 1) => error

last pair [Function]

[R7RS list] Returns the last element of the non-empty, finite list pair. It is equivalent to (car (last-pair pair)).

(last '(1 2 3)) => 3
(last '(1 2 . 3)) => 2
split-at x i

split-at! x i

[Function]

/split-at/ splits the list x at index i, returning a list of the first i elements, and the remaining tail.

(split-at '(a b c d e) 2) ⇒ (a b) (c d e)

split-at! is the linear-update variant. It may destructively modifies x to produce the result.

split-at* list k :optional (fill? #f) (padding #f)

[Function]

More tolerant version of split-at. Returns the results of take* and drop*.

(split-at* '(a b c d) 6 #t 'z)
⇒ (a b c d z z) and ()

slices list k :optional fill? padding

[Function]

Splits list into the sublists (slices) where the length of each slice is k. If the length of list is not a multiple of k, the last slice is dealt in the same way as take*; that is, it is shorter than k by default, or added padding if fill? is true.

(slices '(a b c d e f g) 3)
⇒ ((a b c) (d e f) (g))

(slices '(a b c d e f g) 3 #t 'z)
⇒ ((a b c) (d e f) (g z z))

intersperse item list

[Function]

Inserts item between elements in the list. (The order of arguments is taken from Haskell's intersperse).

(intersperse '+ '(1 2 3)) ⇒ (1 + 2 + 3)
(intersperse '+ '(1)) ⇒ (1)
(intersperse '+ '()) ⇒ ()

6.6.6 Walking over lists

map proc list1 list2 . . .

[Function]

/Map/ applies proc for each element(s) of given list(s), and returns a list of the results. R7RS doesn’t specify the application order of map, but Gauche guarantees proc is always applied in order of the list(s). Gauche’s map also terminates as soon as one of the list is exhausted.

(map car '((a b) (c d) (e f))) ⇒ (a c e)

(map cons '(a b c) '(d e f))
⇒ ((a . d) (b . e) (c . f))

Note that the gauche.collection module (see Section 9.6 [Collection framework], page 404) extends map to work on any type of collection.

append-map f c1ist1 c1ist2 . . .

[Function]

Functionally equivalent to the followings, though a bit more efficient:

(apply append (map f c1ist1 c1ist2 . . .))
(apply append! (map f c1ist1 c1ist2 . . .))

At least one of the list arguments must be finite.
**map* proc tail-proc list1 list2 . . .**  

Like `map`, except that `tail-proc` is applied to the `cdr` of the last pair in the argument(s) to get the `cdr` of the last pair of the result list. This procedure allows improper list to appear in the arguments. If a single list is given, `tail-proc` always receives a non-pair object.

```
(map* - / '(1 2 3 . 4)) ⇒ (-1 -2 -3 . 1/4)
```

```
(define (proper lis)
  (map* values
         (lambda (p) (if (null? p) '() (list p)))
         lis))
```

```
(proper '(1 2 3)) ⇒ (1 2 3)
(proper '(1 2 3 . 4)) ⇒ (1 2 3 4)
```

If more than one list are given, the shortest one determines how `tail-proc` is called. When `map*` reaches the last pair of the shortest list, `tail-proc` is called with cdrs of the current pairs.

```
(map* + vector '(1 2 3 4) '(1 2 . 3))
⇒ (2 4 . #((3 4) 3))
```

Note: The name `map*` is along the line of `list*/cons*` that can produce improper list (See Section 6.6.4 [List constructors], page 142, see Section 10.3.1 [R7RS lists], page 593).

**for-each proc list1 list2 . . .**  

[R7RS base] Applies `proc` for each element(s) of given list(s) in order. The results of `proc` are discarded. The return value of `for-each` is undefined. When more than one list is given, `for-each` terminates as soon as one of the list is exhausted.

Note that the `gauche.collection` module (see Section 9.6 [Collection framework], page 404) extends `for-each` to work on any type of collection.

**fold kons knil clist1 clist2 . . .**  

[R7RS list] The fundamental list iterator. When it is given a single list `clist1 = (e1 e2 . . . en)`, then this procedure returns

```
(kons en . . . (kons e2 (kons e1 knil)) . . .)
```

If `n` list arguments are provided, then the `kons` function must take `n+1` parameters: one element from each list, and the "seed" or fold state, which is initially `knil`. The fold operation terminates when the shortest list runs out of values. At least one of the list arguments must be finite.

Examples:

```
(fold + 0 '(3 1 4 1 5 9)) ⇒ 23 ;sum up the elements
(fold cons () '(a b c d e)) ⇒ (e d c b a) ;reverse
(fold cons* () '(a b c) '(1 2 3 4 5))
⇒ (c 3 b 2 a 1) ;n-ary case
```

**fold-right kons knil clist1 clist2 . . .**  

[R7RS list] The fundamental list recursion operator. When it is given a single list `clist1 = (e1 e2 . . . en)`, then this procedure returns

```
(kons e1 (kons e2 . . . (kons en knil)))
```

If `n` list arguments are provided, then the `kons` function must take `n+1` parameters: one element from each list, and the "seed" or fold state, which is initially `knil`. The fold operation terminates when the shortest list runs out of values. At least one of the list arguments must be finite.
Examples:

\[(\text{fold-right cons '()} '(a b c d e))\]
\[\Rightarrow (a b c d e) \text{ ; copy list}\]

\[(\text{fold-right cons* '()} '(a b c) '(1 2 3 4 5))\]
\[\Rightarrow (a 1 b 2 c 3) \text{ ; n-ary case}\]

**fold-left** snok knil clist1 clist2 . . .

[R6RS] This is another variation of left-associative folding. When it is given a single list clist1 = (e1 e2 . . . en), then this procedure returns:

\[(\text{snok} (\ldots (\text{snok} (\text{snok knil e1) e2) . . .) en)\]

Compare this with fold above; association is the same, but the order of arguments passed to the procedure snok is reversed from the way arguments are passed to kons in fold. If snok is commutative, fold and fold-left produces the same result.

\[(\text{fold-left} + 0 '(1 2 3 4 5) \Rightarrow 15\]

\[(\text{fold-left} \text{ cons 'z} '(a b c d))\]
\[\Rightarrow (((z . a) . b) . c) . d\]

\[(\text{fold-left} (\^ [a b] (\text{cons b a})) 'z '(a b c d))\]
\[\Rightarrow (a b c d z)\]

If more than one lists are given, snok is called with the current seed value knil and each corresponding element of the input lists clist1 clist2 . . . .

\[(\text{fold-left} \text{ list 'z} '(a b c) '(A B C))\]
\[\Rightarrow (((z a A) b B) c C)\]

Note: Most functional languages have left- and right- associative fold operations, which correspond to fold-left and fold-right, respectively. (e.g. Haskell’s foldl and foldr). In Scheme, SRFI-1 first introduced fold and fold-right. R6RS introduced fold-left. (However, in R6RS the behavior is undefined if the lengths of clist1 clist2 . . . aren’t the same, while in Gauche fold-left terminates as soon as any one of the lists terminates.)

**reduce** f ridentity list

**reduce-right** f ridentity list

[R7RS list] Variant of fold and fold-right. f must be a binary operator, and ridentity is the value such that for any value x that is valid as f’s input,

\[(f x \text{ ridentity}) \equiv x\]

These functions effectively do the same thing as fold or fold-right, respectively, but omit the first application of f to ridentity, using the above nature. So ridentity is used only when list is empty.

**filter** pred list

**filter!** pred list

[R7RS list] A procedure pred is applied on each element of list, and a list of elements that pred returned true on it is returned.

\[(\text{filter odd? '}(3 1 4 5 9 2 6)) \Rightarrow (3 1 5 9)\]

filter! is the linear-update variant. It may destructively modifies list to produce the result.

**filter-map** f clist1 clist2 . . .

[R7RS list] Like map, but only true values are saved. At least one of the list arguments must be finite.

\[(\text{filter-map (lambda (x) (and (number? x) (* x x))) '(a 1 b 3 c 7)})\]
\[\Rightarrow (1 9 49)\]
remove pred list
remove! pred list

[R7RS list] A procedure pred is applied on each element of list, and a list of elements that pred returned false on it is returned.

(remove odd? '(3 1 4 5 9 2 6)) ⇒ (4 2 6)

remove! is the linear-update variant. It may destructively modifies list to produce the result.

find pred clist

[R7RS list] Applies pred for each element of clist, from left to right, and returns the first element that pred returns true on. If no element satisfies pred, #f is returned.

find-tail pred clist

[R7RS list] Applies pred for each element of clist, from left to right, and when pred returns a true value, returns the pair whose car is the element. If no element satisfies pred, #f is returned.

any pred clist1 clist2 . . .

[R7RS list] Applies pred across each element of clists, and returns as soon as pred returns a non-false value. The return value of any is the non-false value pred returned. If clists are exhausted before pred returns a non-false value, #f is returned.

every pred clist1 clist2 . . .

[R7RS list] Applies pred across each element of clists, and returns #f as soon as pred returns #f. If all application of pred return a non-false value, every returns the last result of the applications.

count pred clist1 clist2 . . .

[R7RS list] A procedure pred is applied to the n-th element of given lists, from n is zero to the length of the the shortest finite list in the given lists, and the count of times pred returned true is returned.

(count even? '(3 1 4 1 5 9 2 5 6)) ⇒ 3
(count < '(1 2 4 8) '(2 4 6 8 10 12 14 16)) ⇒ 3

At least one of the argument lists must be finite:

(count < '(3 1 4 1) (circular-list 1 10)) ⇒ 2

delete x list :optional elt=
delete! x list :optional elt=

[R7RS list] Equivalent to

(remove (lambda (y) (elt= x y)) list)
(remove! (lambda (y) (elt= x y)) list)

The comparison procedure, elt=, defaults to equal?.

delete-duplicates list :optional elt=
delete-duplicates! list :optional elt=

[R7RS list] Removes duplicate elements from list. If there are multiple equal elements in list, the result list only contains the first or leftmost of these elements in the result. The order of these surviving elements is the same as in the original list. The comparison procedure, elt=, defaults to equal?. 
### 6.6.7 Other list procedures

**append list ...**  
([Function]  
[R7RS base] Returns a list consisting of the elements of the first list followed by the elements of the other lists. The resulting list is always newly allocated, except that it shares structure with the last list argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

**append! list ...**  
([Function]  
[R7RS list] Returns a list consisting of the elements of the first list followed by the elements of the other lists. The cells in the lists except the last one may be reused to construct the result. The last argument may be any object.

**concatenate list-of-lists**  
([Function]  
**concatenate! list-of-lists!**  
([Function]  
[R7RS list] Equivalent to `(apply append list-of-lists)` and `(apply append! list-of-lists!)`, respectively, but this can be a bit efficient by skipping overhead of `apply`.

**reverse list :optional (tail '())**  
([Function]  
**reverse! list :optional (tail '())**  
([Function]  
[R7RS+] Returns a list consisting of the elements of list in the reverse order. While `reverse` always returns a newly allocated list, `reverse!` may reuse the cells of list. Even list is destructively modified by `reverse!`, you should use its return value, for the first cell of list may not be the first cell of the returned list.

If an optional argument `tail` is given, it becomes the tail of the returned list (`tail` isn’t copied). It is useful in the idiom to prepend the processed results on top of already existing results.

\[
\begin{align*}
\text{(reverse } '(1 2 3 4 5)) & \Rightarrow (5 4 3 2 1) \\
\text{(reverse } '(1 2 3) \ '(a b)) & \Rightarrow (3 2 1 a b)
\end{align*}
\]

The `tail` argument is Gauche’s extension, and it isn’t in the traditional Scheme’s `reverse`. The rationale is the following correspondence:

\[
\begin{align*}
\text{(reverse xs)} & \equiv (\text{fold cons xs } '()) \\
\text{(reverse xs tail)} & \equiv (\text{fold cons xs tail})
\end{align*}
\]

**append-reverse rev-head tail**  
([Function]  
**append-reverse! rev-head tail**  
([Function]  
[R7RS list] Equivalent to the two-argument `reverse` and `reverse!`. Provided for SRFI-1 (R7RS `(scheme list)`) compatibility.

**memq obj list**  
([Function]  
**memv obj list**  
([Function]  
**member obj list :optional obj=**  
([Function]  
[R7RS base] Searches obj in the list. If n-th element of list equals to obj (in the sense of `eq?` for `memq`, `eqv?` for `memv`, and `equal?` for `member`), `(list-tail list n)` is returned. Otherwise, `#f` is returned.

If the optional `obj=` argument of `member` is given, it is used as an equivalence predicate instead of `equal?`.

\[
\begin{align*}
\text{(memq 'a } '(a b c)) & \Rightarrow (a b c) \\
\text{(memq 'b } '(a b c)) & \Rightarrow (b c) \\
\text{(memq 'a } '(b c d)) & \Rightarrow #f \\
\text{(memq (list 'a) } '(b (a) c)) & \Rightarrow #f \\
\text{(memv 101 } '(100 101 102)) & \Rightarrow (101 102)
\end{align*}
\]
6.6.8 Association lists

acons obj1 obj2 obj3

[Function] Returns (cons (cons obj1 obj2) obj3). Useful to put an entry at the head of an associative list.

(This procedure is defined in SRFI-1 (R7RS (scheme list)) as alist-cons; see Section 10.3.1 [R7RS lists], page 593).

(acons 'a 'b '((c . d))) ⇒ ((a . b) (c . d))

alist-copy alist

[Function] [R7RS list] Returns a fresh copy of alist. The spine of alist and each cell that points a key and a value is copied.

(define a (list (cons 'a 'b) (cons 'c 'd)))

a ⇒ ((a . b) (c . d))

(define b (alist-copy a))

b ⇒ ((a . b) (c . d))

(set-cdr! (car a) 'z)

a ⇒ ((a . z) (c . d))

b ⇒ ((a . b) (c . d))

assq obj list

assv obj list

assoc obj list :optional key=

[Function] [R7RS base] Each element in list should be a pair (Gauche ignores non-pair element in list, but other R7RS implementation may raise an error, so be aware of it when you’re writing a portable code). These procedures search a pair whose car matches obj (in the sense of eq? for assq, eqv? for assv, and equal? for assoc) from left to right, and return the leftmost matched pair if any. If no pair matches, these return #f.

If the optional argument of assoc is given, it is called instead of equal? to check the equivalence of obj and each key.

alist-delete key alist :optional key=

alist-delete! key alist :optional key=

[Function] [R7RS list] Deletes all cells in alist whose key is the same as key. Comparison is done by a procedure key=. The default is eqv?.

The linear-update version alist-delete! may or may not modify alist.

rassoc key alist :optional eq-fn

rassq key alist

rassv key alist

[Function] Reverse associations—given key is matched to the cdr of each element in alist, instead of the car. Handy to realize bidirectional associative list. Rassoc takes an optional comparison function, whose default is equal?. Rassq and rassv uses eq? and eqv?.

assoc-ref alist key :optional default eq-fn

assq-ref alist key :optional default

assv-ref alist key :optional default

[Function] These procedures provide the access to the assoc list symmetric with other *-ref procedures. (Note that the argument order is different from assoc, assq and assv – *-ref procedures take a container first, and an item second.)
This captures the common pattern of alist access:

\[
\text{(assoc-ref alist key default eq-fn)} = \\
\text{(cond \[(assoc key alist eq-fn) => cdr]\ [else default]])}
\]

If \text{default} is omitted, \text{#f} is used.

\text{Assoc-ref} takes an optional comparison function \text{eq-fn}, whose default is \text{equal?}. \text{Assq-ref} and \text{assv-ref} uses \text{eq?} and \text{eqv?}, respectively.

\text{rassoc-ref} \text{alist key :optional default eq-fn}

\text{rassq-ref} \text{alist key :optional default}

\text{rassv-ref} \text{alist key :optional default}

Reverse association version of \text{assoc-ref}.

\[
\text{(rassoc-ref alist key default eq-fn)} = \\
\text{(cond \[(rassoc key alist eq-fn) => car]\ [else default]])}
\]

The meanings of optional arguments are the same as \text{assoc-ref}.

\text{assoc-set!} \text{alist key val :optional eq-fn}

\text{assq-set!} \text{alist key val}

\text{assv-set!} \text{alist key val}

Returns an alist who has \text{(key . val)} pair added to the \text{alist}. If \text{alist} already has an element with \text{key}, the element’s \text{cdr} is destructively modified for \text{val}. If \text{alist} doesn’t have an element with \text{key}, a new pair is created and appended in front of \text{alist}; so you should use the return value to guarantee \text{key-val} pair is added.

\text{Assoc-set!} takes optional comparison function \text{eq-fn}, whose default is \text{equal?}. \text{Assq-set!} and \text{assv-set!} uses \text{eq?} and \text{eqv?}, respectively.

\text{assoc-adjoin} \text{alist key val :optional eq-fn}

If \text{alist} contains an entry with \text{key}, returns a new associative list where the value of the \text{key} is replaced for \text{val}. The order of entries in \text{alist} is preserved. If \text{alist} doesn’t contain the entry, it returns \text{acons key val alist}.

The original \text{alist} is left unmodified. The returned associative list may share a part of its tail with the original \text{alist}, however.

The optional \text{eq-fn} argument is a procedure with two arguments to be used to compare the keys; the default is \text{equal?}.

Note the order of arguments; we have \text{alist} first, just as \text{assoc-ref} and \text{assoc-set!}, and other \text{-adjoin} procedures. It is not the same as \text{alist-delete} and \text{assoc}, which takes the key first.

\text{assoc-update-in} \text{alist keys proc :optional default eq-fn}

This procedure allows to update a nested associative list. The \text{alist} argument is a (possibly nested) associative list, \text{keys} are a list of keys, and \text{proc} is a procedure that takes one argument. First, the keys are looked up recursively in \text{alist}; then its value is passed to \text{proc}. The return value is a new (nested) associative list where the value pointed by \text{keys} is replaced with the return value of \text{proc}.

\[
\text{(assoc-update-in '((a (b . 1) (c . 2)) (a c) (cut + <> 1)))} =,(a b . 1) c . 3)
\]

The order of entries are preserved. The original \text{alist} is left unmodified, but the returned value may share a part of the structure with \text{alist}.
If `alist` doesn’t have the entry specified by `keys`, a new entry is added. A new entry is added at the beginning of the sequence where specified key didn’t exist.

```
(assoc-update-in '(((a (b . 1) (c . 2))) '(a d e) (^_ 99))
  ⇒ ((a (d (e . 99)) (b . 1) (c . 3)))
```

The `default` argument is passed to `proc` when there’s no entry with specified keys. If omitted, `#f` is assumed.

The optional `eq-fn` argument is a procedure with two arguments to be used to compare the keys; the default is `equal?`.

Note the order of arguments; we have `alist` first, just as `assoc-ref` and `assoc-set!`, and other `-adjoin` procedures. It is not the same as `alist-delete` and `assoc`, which takes the key first.

Note: For destructively updating general nested aggregate structures, setter of `~` is handy (see Section 6.15.2 [Universal accessor], page 219). You can modify an entry in a hashtable in a vector in a list, for example. Associative list is a bit special, since you can’t distinguish it from lists (thus `~` can’t be used), and it is mostly used in functional way. So we added a special update procedure.

### 6.6.9 Extended pairs and pair attributes

Gauche has a special kind of pairs, called extended pairs. It behaves exactly the same as ordinary pairs, but you can associate an attribute list to it. Gauche uses it to keep source-code location information, for example.

Extended pairs don’t incur any overhead in accessing its car/cdr; `set-car!` and `set-cdr!` has a little overhead (another reason you should avoid mutation!). Internally it takes up twice of memory than the ordinary pairs.

Pair attributes are not written out in the external representation, and not copied by `copy-list` and many other list procedures that constructs the output list from the input list. That is, they are easily lost. Use them for the ancillary information which is nice to have but not mandatory.

Keep in mind that code using extended pairs is not easily ported to other Scheme implementations, although the feature can be emulated with a separate weak hash table.

**extended-pair? obj**

Returns `#t` iff `obj` is an extended pair.

**extended-cons car cdr :optional attrs**

Returns an extended pair of `car` and `cdr`. If an optional `attrs` argument is given, it must be an alist, specifying initial pair attributes. By default, the pair attributes of the created extended pair is empty.

**extended-list obj obj2 ...**

Creates and returns a list of `obj obj2 ...`, but its first pair is an extended pair. Note that the subsequent pairs are ordinary pairs.

**pair-attributes pair**

Returns pair attributes of `pair` as an alist. You can pass an ordinary pair, in which case an empty list is returned.

```
(pair-attributes (extended-cons 'a 'b '((c . d) (e . f))))
  ⇒ ((c . d) (e . f))
```

```
(pair-attributes (cons 'a 'b))
  ⇒ ()
```
**pair-attribute-get** `pair key :optional default`  
[Function]  
Returns the value associated to the key `key` in the pair attributes of `pair`. `Key` can be any Scheme object, and compared with `eq?`. If there’s no value associated with the given key, `default` is returned if it is given, otherwise an error is signaled.  
You can pass an ordinary pair as `pair`; in that case, it is treated with empty pair attributes.

**pair-attribute-set!** `pair key value`  
[Function]  
Adds a pair attribute of `key` with `value` to an extended pair `pair`. `Key` and `value` can be any Scheme object. An error is thrown if `pair` is not an extended pair.  
This procedure does not mutate the exising alist, but rather makes a necessary copy. The pair attributes are not supposed to be mutated frequently.

### 6.7 Symbols

**<symbol>**  
[Builtin Class]  
A class for symbols.

**|name|**  
[Reader Syntax]  
[R7RS] Denotes a symbol that has weird name, including the characters that are not usually allowed in symbols. It can also include hex-escaped characters.

```
;; A symbol with spaces in its name
'|this is a symbol| ⇒ |this is a symbol|
```

```
;; Unicode codepoint can be used following backslash-x escape,;; and terminated by semicolon.
'|\x3bb;| ⇒ λ
```

If the interpreter is running in case-insensitive mode, this syntax can be used to include uppercase characters in a symbol (see Section 2.4 [Case-sensitivity], page 15).

**#:name**  
[Reader Syntax]  
Denotes uninterned symbol. Uninterned symbols can be created by `gensym` or `string->uninterned-symbol`.

Uninterned symbols are mainly for legacy macros to avoid variable conflicts. They are not registered in the internal dictionary, so such symbols with the same name can’t be `eq?`.

```
(eq? '#:foo '#:foo) ⇒ #f
(eq? '#:foo 'foo) ⇒ #f
```

To preserve `eq?-ness` of uninterened symbols, you need to use `write-shared` which shows shared structures.

```
(write-shared (let1 s '#:foo (list s s)))
⇒ prints (#0=#:foo #0#)
```

```
(write-shared (let ((s '#:foo) (t '#:foo)) (list s t s t)))
⇒ prints (#0=#:foo #1=#:foo #0# #1#)
```

**symbol?** `obj`  
[R7RS base] Returns true if and only if `obj` is a symbol.

```
(symbol? ’abc) ⇒ #t
(symbol? 0) ⇒ #f
(symbol? ’i) ⇒ #t
(symbol? ’-i) ⇒ #f
(symbol? ’|-i|) ⇒ #t
```
symbol-interned? symbol
Returns #t if symbol is an interned symbol, #f if it is an uninterned symbol. An error is signaled if symbol is not a symbol.

symbol=? a b c ...
[R7RS base] Every argument must be a symbol. Returns #t iff every pair of arguments are eq? to each other.

symbol->string symbol
[R7RS base] Returns the name of symbol in a string. Returned string is immutable.
(symbol->string 'foo) ⇒ foo

string->symbol string
[R7RS base] Returns a symbol whose name is a string string. String may contain weird characters.
(string->symbol "a") ⇒ a
(string->symbol "A") ⇒ A
(string->symbol "weird symbol name") ⇒ |weird symbol name|

string->uninterned-symbol string
Like string->symbol, but the created symbol is uninterned.
(string->uninterned-symbol "a") ⇒ #:a

gensym :optional prefix
Returns a fresh, uninterned symbol. The returned symbol can never be eq? to other symbol within the process. If prefix is given, which must be a string, it is used as a prefix of the name of the generated symbol. It is mainly for the convenience of debugging.

symbol-sans-prefix symbol prefix
Both symbol and prefix must be symbols. If the name of prefix matches the beginning part of the name of symbol, this procedure returns a symbol whose name is the name of symbol without the matched prefix. Otherwise, it returns #f.
(symbol-sans-prefix 'foo:bar 'foo:) ⇒ bar
(symbol-sans-prefix 'foo:bar 'baz:) ⇒ #f

symbol-append interned? objs ...
(symbol-append obj ...)
Returns a symbol with the name which is a concatenation of string representation of objs.

If the first argument is a boolean, it is recognized as the first form; the first argument specifies whether the resulting symbol is interned or not.

Each other argument is converted to a string as follows: If it is a keyword, its name (with the preceding : ) is used. For all other objects, x->string is used. (The special treatment of keyword is to keep the consistency before and after keyword-symbol integration. See Section 6.8.1 [Keyword and symbol integration], page 158, for the details.)

This is upper-compatible to Bigloo's same name procedure, which only allows symbols as the arguments and the result is always interned.
(symbol-append 'ab 'cd) ⇒ abcd
(symbol-append 'ab ':c 30) ⇒ ab:c30
(symbol-append #f ':g 100) ⇒ #:g100
6.8 Keywords

Keywords are a subtype of symbols that are automatically bound to itself. It is extensively used in named arguments (keyword arguments), and keyword-value list.

See Section 4.3 [Making procedures], page 47, for how Gauche supports keyword arguments, and let-keywords macro (Section 6.15.4 [Optional argument parsing], page 222) for parsing keyword-value list manually.

Keywords used to be a disjoint type from symbols. Since it isn’t conformant to R7RS, in which symbols can begin with :, we’ve introduced two modes since 0.9.5; keywords can be a disjoint type of its own, or it can be a subtype of symbols.

The behavior can be switched by environment variables. If the environment variable GAUCHE_KEYDISJOINT is defined when gosh starts up, keywords and symbols are disjoint. Otherwise, if the environment variable GAUCHE_KEYIS_SYMBOL is defined, keywords are a subtype of symbols.

The default behavior when neither environment variables are defined has been switched since 0.9.8. GAUCHE_KEYDISJOINT was assumed in 0.9.7 and before, while GAUCHE_KEYIS_SYMBOL is assumed in 0.9.8 and after.

Most typical code run in either mode, but there can be some code that behaves differently. See Section 6.8.1 [Keyword and symbol integration], page 158, for effect of the change.

In future we’ll stop supporting GAUCHE_KEYDISJOINT, so we recommend you to ensure applications to run on the current default mode.

: name

[Reader Syntax]

Read to a keyword whose name is : name.

keyword? obj

[Function]

Returns #t if obj is a keyword.

make-keyword name

[Function]

Returns a keyword whose name is name prepended by :. The name argument can be a string or a symbol.

(make-keyword "foo") ⇒ :foo

(make-keyword 'foo) ⇒ :foo

keyword->string keyword

[Function]

Returns the name (without the initial :) of the keyword keyword, in a string.

(keyword->string :foo) ⇒ "foo"

get-keyword key kv-list :optional fallback

[Function]

A useful procedure to extract a value from key-value list. A key-value list kv-list must contains even number of elements; the first, third, fifth ... elements are regarded as keys, and the second, fourth, sixth ... elements are the values of the preceding keys.

This procedure looks for key from the keys, and if it finds one, it returns the corresponding value. If there are more than one matching keys, the leftmost one is taken. If there is no matching key, it returns fallback if provided, or signals an error otherwise.

It is an error if kv-list is not a proper, even-number element list.

Actually, ‘keywords’ in the keyword-value list and the key argument need not be a keyword—it can be any Scheme object. Key comparison is done by eq?.

This procedure is taken from STk.

\[
\begin{align*}
\text{(get-keyword } :y \ '(:x \ 1 \ :y \ 2 \ :z \ 3)) & \Rightarrow 2 \\
\text{(get-keyword } 'z \ '(:x \ 1 \ :y \ 2 \ :z \ 3)) & \Rightarrow 3 \\
\text{(get-keyword } :t \ '(:x \ 1 \ :y \ 2 \ :z \ 3)) & \Rightarrow \#<\text{error}> \\
\text{(get-keyword } :t \ '(:x \ 1 \ :y \ 2 \ :z \ 3) \ #f) & \Rightarrow #f
\end{align*}
\]

\text{get-keyword* key kv-list :optional fallback} \quad \text{[Macro]}

Like \text{get-keyword}, but \text{fallback} is evaluated only if \text{kv-list} does not have \text{key}.

\text{delete-keyword key kv-list} \quad \text{[Function]}
\text{delete-keyword! key kv-list} \quad \text{[Function]}

Removes all the keys and values from \text{kv-list} for keys that are \text{eq?} to \text{key}.

\text{delete-keyword} doesn’t change \text{kv-list}, but the returned list may share the common tail of it.

\text{delete-keyword!} doesn’t allocate, and \text{may} destructively changes \text{kv-list}. You still have to use the returned value, for the original list may not be changed if its first key matches \text{key}.

If there’s no key that matches \text{key}, \text{kv-list} is returned.

\[
\text{(delete-keyword } :y \ '(:x \ 1 \ :y \ 2 \ :z \ 3 \ :y \ 4)) \Rightarrow (:x \ 1 \ :z \ 3)
\]

\text{delete-keywords keys kv-list} \quad \text{[Function]}
\text{delete-keywords! keys kv-list} \quad \text{[Function]}

Similar to \text{delete-keyword} and \text{delete-keyword!}, but you can specify a list of objects in \text{keys}; when a key in \text{kv-list} matches any of \text{keys}, the key and the following value is removed from \text{kv-list}.

\[
\text{(delete-keywords } '(:x \ :y) \ '(:x \ 1 \ :y \ 2 \ :z \ 3 \ :y \ 4)) \Rightarrow (:z \ 3)
\]

\section{6.8.1 Keyword and symbol integration}

In older versions of Gauche, keywords are of disjoint type from symbols, and they are self-evaluating objects. To maintain the compatibility, the current Gauche makes symbols that begins with : automatically bound to itself.

On the surface it won’t make much difference; you can write a keyword :key, which evaluates to itself; so you can pass and receive keyword arguments just as they used to be. If you use :key as variables, however, e.g. \text{(define :key 3)}, the value of :key in your module changes (it won’t affect other modules, which refer to the binding of :key in \text{gauche.keyword} module).

However, there are several subtle points that do make difference, that breaks compatibility of legacy code. We explain here how to change the code that works in both ways.

If you find a problem in new mode and want to get the old behavior until you change the code, you can set the environment variable \text{GAUCHE_KEYWORD_DISJOINT}.

\text{(symbol? :key)} used to return \#f, now returns \#t

\text{keyword?} always returns \#t on keywords, but if you need to switch behavior depending whether an object is a symbol or a keyword, you should test keyword-ness first.

\text{;; behaved differently in 0.9.7 and before}
(cond
  [(symbol? x) (x-is-symbol)]
  [(keyword? x) (x-is-keyword)])

;; works on all versions
(cond
  [(keyword? x) (x-is-keyword)]
  [(symbol? x) (x-is-symbol)])

**Literal keywords in pattern matching**

In the old versions, when keywords appear in a pattern of `util.match` or `syntax-rules`, they only matched to themselves. In the current version, such keywords in a pattern are treated as pattern variables, since they are symbols.

;; In the old versions
(match '(a b) [(:key z) (list :key z)] [_ "nope"])
⇒ "nope"

;; In the current version
;; :key is treated just as a pattern variable
(match '(a b) [(:key z) (list :key z)] [_ "nope"])
⇒ (a b)

The same thing happens to the patterns in `syntax-rules`.

To make the code work in both versions, explicitly mark the keywords as literals.

- For `match`, quote the keywords you want to be treated as literals.
  (match '(a b) [('':key z) (list :key z)] [_ "nope"])
  ⇒ "nope"

- For `syntax-rules`, list the keywords as literals.
  (syntax-rules (:key)
    [(_:key z) (list :key z)]) ;etc.

As of Gauche 0.9.5, `match` warns if you have unquoted keywords in match patterns.

**Displaying keywords**

(display :key) used to print key (no colon), while it now prints :key.

You can use `(display (keyword->string :key))` which prints key in both versions.

**For R7RS code, quote them or import Gauche modules**

Keywords (symbols beginning with :) are automatically bound to itself in the `gauche.keyword` module.

Gauche code inherits the `gauche` module by default, which inherits `keyword`, so you can see the binding of the keyword by default.

In R7RS code, however, you don’t inherit `gauche`, so symbols beginning with : are just ordinary symbols by default. Usually you do `(import (gauche base))` to use Gauche built-ins, and that makes binding of `gauche.keyword` available in your code, too (since `gauche.base` inherits `gauche.keyword`). But keep this in mind just in case you want to handle keywords in your R7RS code separate from Gauche procedures—you have to either say `(import (gauche keyword))` to get just the self-bound keywords, or quote them.

(import (scheme base))
:foo ⇒ ERROR: unbound variable: :foo

(import (gauche base))

:foo ⇒ :foo

In the following example, the R7RS library foo imports only copy-port from (gauche base); in that case, you have to import (gauche keyword) separately in order to use :size keyword without quoting. (Or add :size explicitly in the imported symbol list of (gauche base).)

(define-library (foo)
  (import (scheme base)
    (only (gauche base) copy-port)
    (gauche keyword))

(export cat)

(begin
  (define (cat)
    (copy-port (current-input-port)
      (current-output-port)
      :size 4096))))

6.9 Characters

<char> [Builtin Class]

#\charname [Reader Syntax]

[R7RS+] Denotes a literal character.

When the reader reads #\, it fetches a subsequent character. If it is one of () [] {} " \ ; #, this is a character literal of itself. Otherwise, the reader reads subsequent characters until it sees a non word-constituent character. If only one character is read, it is the character. Otherwise, the reader matches the read characters with predefined character names. If it doesn’t match any, an error is signaled.

The following character names are recognized. These character names are case insensitive.

space Whitespace (ASCII #x20)
newline, nl, lf
  Newline (ASCII #x0a)
return, cr
  Carriage return (ASCII #x0d)
tab, ht
  Horizontal tab (ASCII #x09)
page
  Form feed (ASCII #x09)
alarm
  Bell (ASCII #x07)
backspace
  Backspace (ASCII #x08)
escape, esc
  Escape (ASCII #x1b)
delete, del
  Delete (ASCII #x7f)
nul
  NUL character (ASCII #x00)
A character whose Unicode codepoint is the integer \(N\), when \(N\) is a hexadecimal integer. This is R7RS lexical syntax. (See the compatibility note below).

A character whose Unicode codepoint is the integer \(N\), where \(N\) is a 4-digit or 8-digit hexadecimal number. This is legacy Gauche lexical syntax. Use \(\text{xN}\) syntax for the new code. (See the compatibility note below).

---

**Compatibility note:** Before 0.9.4, \(\text{xNN}\) syntax uses Gauche’s internal character encoding as opposed to Unicode codepoint. Both are the same if Gauche is compiled with internal encoding `utf-8` or `none` (if it’s `none`, only characters up to U+00ff is supported and in this range the characters are the same as Unicode characters.) If Gauche is compiled with encoding `euc-jp` or `sjis`, the meaning of \(\text{xNN}\) beyond ASCII range differs from 0.9.3.3 or before.

If you set the reader mode to `legacy` (see Section 6.21.7.2 [Reader lexical mode], page 272), \(\text{\#\text{xNN}}\) is read as before, keeping the compatibility (but it isn’t compatible to R7RS). Alternatively, you can use \(\text{\#\text{uNNNN}}\), or a character itself, to make the code work in both new and old versions of Gauche.

### Function

`char? obj` [Function]

[R7RS base] Returns `#t` if `obj` is a character, `#f` otherwise.

`char=? char1 char2 char3 . . .` [Function]

`char<? char1 char2 char3 . . .` [Function]

`char<? char1 char2 char3 . . .` [Function]

`char>? char1 char2 char3 . . .` [Function]

`char>=? char1 char2 char3 . . .` [Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

[Function]

### R7RS char

compares characters. Character comparison is done in internal character encoding.

### char-ci=? char1 char2 char3 . . .

### char-ci<? char1 char2 char3 . . .

### char-ci<=? char1 char2 char3 . . .

### char-ci>? char1 char2 char3 . . .

### char-ci>=? char1 char2 char3 . . .

[Function]

### R7RS char]

compares characters in case-insensitive way. The comparison is done in the internal character code of the foldcase of the each character; see `char-foldcase` below.

In R7RS, these procedures are in the `(scheme char)` library.

### char-alphabetic? char

### char-numeric? char

### char-whitespace? char

### char-upper-case? char

### char-lower-case? char

### char-title-case? char

[R7RS char][SRFI-129] Returns true if a character `char` is an alphabetic character (Unicode character category Lu, Ll, Lt, Lm, Lo, Nl), a numeric character (Unicode character category Nd), a whitespace character, (Unicode character category Zs, Zp, Zl), an upper case character (Unicode character category Zu), or a lower case character (Unicode character category Ll), respectively.
In R7RS, these procedures except `char-title-case?` are in the `(scheme char)` library, while `char-title-case?` is defined in SRFI-129.

`char-word-constituent?  char`  
Returns `#t` if `char` is a character R7RS allows to be in an identifier name without escaping: ASCII alphanumeric characters, extended alphabetic characters (`! $ % & * + - . / : < = > ? ^ _ ~`), any characters whose category is Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co, Zero width non-joiner (U+200C), and Zero width joiner (U+200D).

`char-general-category  char`  
[R6RS] Returns one of the following symbols, representing the Unicode general category of `char`.

- Cc: Other, Control
- Cf: Other, Format
- Cn: Other, Not Assigned
- Co: Other, Private Use
- Cs: Other, Surrogate
- Ll: Letter, Lowercase
- Lm: Letter, Modifier
- Lo: Letter, Other
- Lt: Letter, Titlecase
- Lu: Letter, Uppercase
- Mc: Mark, Spacing Combining
- Me: Mark, Enclosing
- Mn: Mark, Nonspacing
- Nd: Number, Decimal Digit
- Nl: Number, Letter
- No: Number, Other
- Pc: Punctuation, Connector
- Pd: Punctuation, Dash
- Pe: Punctuation, Close
- Pf: Punctuation, Final quote
- Pi: Punctuation, Initial quote
- Po: Punctuation, Other
- Ps: Punctuation, Open
- Sc: Symbol, Currency
- Sk: Symbol, Modifier
- Sm: Symbol, Math
- So: Symbol, Other
- Zl: Separator, Line
- Zp: Separator, Paragraph
- Zs: Separator, Space

If Gauche is compiled with euc-jp or shift_jis encoding, there are characters that don’t have corresponding Unicode codepoint (each of them are represented by one unicode character plus one unicode modifier character). A provisional category is assigned to those characters. If future versions of Unicode incorporates these characters, the category may be reassigned.

<table>
<thead>
<tr>
<th>SJIS</th>
<th>EUC</th>
<th>Cat</th>
<th>Unicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>82F5</td>
<td>A4F7</td>
<td>Lo</td>
<td>U+304B U+309A (Semi-voiced Hiragana KA)</td>
</tr>
<tr>
<td>82F6</td>
<td>A4F8</td>
<td>Lo</td>
<td>U+304D U+309A (Semi-voiced Hiragana KI)</td>
</tr>
<tr>
<td>82F7</td>
<td>A4F9</td>
<td>Lo</td>
<td>U+304F U+309A (Semi-voiced Hiragana KU)</td>
</tr>
<tr>
<td>82F8</td>
<td>A4FA</td>
<td>Lo</td>
<td>U+3051 U+309A (Semi-voiced Hiragana KE)</td>
</tr>
<tr>
<td>82F9</td>
<td>A4FB</td>
<td>Lo</td>
<td>U+3053 U+309A (Semi-voiced Hiragana KO)</td>
</tr>
</tbody>
</table>
The result is undefined if you pass \( n \) to integer->char that doesn’t have a corresponding character.

Converting a character \( \text{char} \) to integer UCS codepoint, and integer UCS codepoint \( \text{n} \) to a character, respectively.

If Gauche is compiled with UTF-8 encoding, these procedures are the same as char->integer and integer->char.

When Gauche’s internal encoding differs from UTF-8, these procedures implicitly loads gauche.charconv module to convert internal character code to UCS or vice versa (see Section 9.5 [Character code conversion], page 399). If char doesn’t have corresponding UCS codepoint, char->ucs returns #f. If UCS codepoint \( n \) can’t be represented in the internal character encoding, ucs->char returns #f, unless the conversion routine provides a substitution character.

char-upcase char
char-downcase char
char-titlecase char
char-foldcase char

[Function]
[R7RS char][SRFI-129] Returns the upper case, lower case, title case and folded case of char, respectively.

The mapping is done according to Unicode-defined character-by-character case mapping whenever possible. If the native encoding doesn’t support the mapped character defined in Unicode, the operation becomes no-op. If the native encoding is ‘none’, we treat the characters as if they are Latin-1 (ISO-8859-1) characters. So, upcasing Latin-1 character small y with diaresis (U+00ff) maps to capital y with diacresis (U+0178) if the internal encoding is utf-8, but it is no-op if the internal encoding is none.

R7RS defines char-upcase, char-downcase, and char-foldcase in the (scheme char) library, while char-titlecase is defined in SRFI-129. R6RS defines all of them.

The character-by-character case mapping doesn’t consider a character that may map to more than one characters; a notable example is eszett (latin small letter sharp S, U+00df), which is is mapped to two capital S’s in string context, but char-upcase #\ß returns #\ß. To get a full mapping, use string-upcase etc. in gauche.unicode module (see Section 9.37.3 [Full string case conversion], page 554).

digit->integer char :optional (radix 10) (extended-range? #f)

[Function]
If given character char is a valid digit character in radix radix number, the corresponding integer is returned. Otherwise #f is returned.

(digit->integer #\4) ⇒ 4  
(digit->integer #\e 16) ⇒ 14  
(digit->integer #\9 8) ⇒ #f

If the optional extended-range? argument is true, this procedure recognizes not only ASCII digits, but also all characters with Nd general category—such as FULLWIDTH DIGIT ZERO to NINE (U+ff10 - U+ff19).

R7RS has digit-value, which is equivalent to (digit->integer char 10 #t).

Note: CommonLisp has a similar function in rather confusing name, digit-char-p.

integer->digit integer :optional (radix 10) (basechar1 #\0) (basechar2 #\a)

[Function]
Reverse operation of digit->integer. Returns a character that represents the number integer in the radix radix system. If integer is out of the valid range, #f is returned.

(integer->digit 13 16) ⇒ #\d  
(integer->digit 10) ⇒ #f

The optional basechar1 argument specifies the character that stands for zero; by default, it’s #\0. You can give alternative character, for example, U+0660 (ARABIC-INDIC DIGIT ZERO) to convert an integer to a arabic-indic digit character.

Another optional basechar2 argument is used for integers over 10. The default value is #\a. You can pass #\A to get upper-case hex digits, for example.

Note: CommonLisp’s digit-char.

gauche-character-encoding

[Function]
Returns a symbol designates the native character encoding, selected at the compile time. The possible return values are those:

euc-jp    EUC-JP
utf-8     UTF-8
sjis      Shift JIS
none      No multibyte character support (8-bit fixed-length character).
To switch code at compile time according to the internal encoding, you can use feature identifiers `gauche.ces.*`—see Section 3.5 [Platform-dependent features], page 34.

**supported-character-encodings**

*Function*

Returns a list of string names of character encoding schemes that are supported in the native multibyte encoding scheme.

### 6.10 Character Sets

**<char-set>**

*Built-in Class*

Character set class. Character set object represents a set of characters. Gauche provides built-in support of character set creation and a predicate that tests whether a character is in the set or not.

The class implements the collection protocol (see Section 9.6 [Collection framework], page 404), so that the standard collection methods provided in the `gauche.collection` module can be used.

An instance of `<char-set>` is applicable to a character, and works as a membership predicate; see `char-set-contains?` below.

Further operations, such as set algebra, is defined in SRFI-14 module (see Section 10.3.6 [R7RS character sets], page 614).

#### 6.10.1 Character set literals

**#[char-set-spec]**

*Reader Syntax*

You can write a literal character set in this syntax. `char-set-spec` is a sequence of characters to be included in the set. You can include the following special sequences:

- **x-y**
  Characters between `x` and `y`, inclusive. `x` must be smaller than `y` in the internal encoding.

- **^**
  If `char-set-spec` begins with caret, the actual character set is a complement of what the rest of `char-set-spec` indicates.

- **\xN**;
  A character whose Unicode codepoint is a hexadecimal number `N`.

- **\uXXXX**
  **\UXXXXX**
  This is a legacy Gauche syntax, for a unidec character whose Unicode codepoint is represented by 4-digit and 8-digit hexadecimal numbers, respectively.

- **\s**

- **\S**
  Complement of whitespace characters.

- **\d**

- **\D**
  Complement of decimal digit characters.

- **\w**
  Word constituent characters (#[A-Za-z0-9_]). Members of `char-set:ascii-word`.

- **\W**
  Complement of word constituent characters.

- **\\**
  A backslash character.

- **\-**
  A minus character.

- **\^**
  A caret character.
[:,alnum:] ... Character set a la POSIX. See the table below for the complete list of recognized character set names. The set name must be in all lower cases. This notation only includes characters in ASCII range.

[[:^alnum:]] ... Complement set of [:alnum:] etc.

[[:ALNUM:]] ... Gauche’s extension of character set a la POSIX; the name must be all in upper cases, and includes full Unicode range. See the table below for the recognized names.

[[:^ALNUM:]] ... Complement set of [:ALNUM:] etc.

Here’s the list of POSIX-style character class names:

- :cntrl: ASCII control characters. char-set:ascii-control, U+0000 to U+001f and U+007f.
- :digit: ASCII digits. char-set:ascii-digit, #[0-9]
- :lower: ASCII lower-case alphabets. char-set:ascii-lower-case, #[a-z]
- :word: ASCII word characters (not POSIX). char-set:ascii-word, #[0-9A-Za-z_]
- :xdigit: Hexadecimal digits. char-set:hex-digit, #[0-9a-fA-F]
- :WORD: Unicode word characters. char-set:word.
- :XDIGIT: Hexadecimal digits (same as :xdigit:).

Here are some examples:

# [aeiou] ; a character set consists of vowels
# [a-zA-Z] ; alphabet
Literal character sets are immutable, as other literal data. An error is signalled when you attempt to modify an immutable character set.

**Note for the compatibility:** We used to recognize a syntax \xNN (two-digit hexadecimal number, without semicolon terminator) as a character; for example, #\x0d\x0a as a return and a newline. We still support it when we don’t see the terminating semicolon, for the compatibility. There are ambiguous cases: #\x0a; means only a newline in the current syntax, but a newline and a semicolon in legacy syntax.

Setting the reader mode to `legacy` restores the old behavior. Setting the reader mode to `warn-legacy` makes it work like the default behavior, but prints warning when it finds legacy syntax. See Section 6.21.7.2 [Reader lexical mode], page 272, for the details.

To write code that can work both in new and old syntax, use \u escape.

### 6.10.2 Predefined character sets

We provide a bunch of predefined character sets, including the ones defined in R7RS charset library (see Section 10.3.6 [R7RS character sets], page 614). Those character sets are immutable.

```scheme
(char-set:letter) ; [R7RS charset] Letters (Unicode general category Lu, Ll, Lt, Lm and Lo).
(char-set:lower-case) ; [Variable]
(char-set:upper-case) ; [Variable]
(char-set:title-case) ; [Variable]
  [R7RS charset] Lower case, upper case and title case letters (Unicode general category Ll, Lu and Lt, respectively).
(char-set:digit) ; [R7RS charset] Digit characters (Unicode general category Nd). Note that this contains many more characters than ASCII 0 to 9. If you need #[0-9], use char-set:ascii-digit.
(char-set:hex-digit) ; [Variable]
  [R7RS charset] Digit characters used for hexadecimal, i.e. #[0-9A-Fa-f]. This does not contain other Unicode digit characters, for it isn’t practical to mix non-ascii digit characters with hexadecimal notation.
(char-set:letter+digit) ; [Variable]
(char-set:graphic) ; [Variable]
  [R7RS charset] Characters that has some glyph. Union of letters, numbers, punctuations and symbols.
(char-set:printing) ; [Variable]
(char-set:whitespace) ; [Variable]
(char-set:blank) ; [Variable]
  [R7RS charset] Whitespace and blank characters; char-set:whitespace includes \tab, \newline, \u000B (vertical tab), \page, \return, and all characters in general category Zs, Zl, Zp, while char-set:blank includes \tab and all characters in general category Zs. Note that char-set:whitespace is the same set of characters that Scheme reader treats as whitespace characters.
```
Chapter 6: Core library

char-set:iso-control
[Variable]
[R7RS charset] Control characters (Unicode general category Cc).

char-set:punctuation
[Variable]
[R7RS charset] Punctuation characters (Unicode general category Pc, Pd, Ps, Pe, Pi, Pf and Po).

char-set:symbol
[Variable]
[R7RS charset] Symbol characters (Unicode general category Sm, Sc, Sk and So).

char-set:ascii
[Variable]
[R7RS charset] Contains all ASCII characters (U+0000 to U+007f).

char-set:empty
[Variable]
[R7RS charset] An empty character set.

char-set:full
[Variable]
[R7RS charset] A character set that includes all characters.

char-set:word
[Variable]
A word constituent characters. In the current version, it is equivalent to char-set:ascii-word (#[0-9A-Za-z_]) but in future versions we may extend this to other Unicode characters. If you intend to mean ASCII-only words, use char-set:ascii-word.

char-set:ascii-letter
[Variable]
char-set:ascii-lower-case
[Variable]
char-set:ascii-upper-case
[Variable]
char-set:ascii-digit
[Variable]
char-set:ascii-letter+digit
[Variable]
char-set:ascii-graphic
[Variable]
char-set:ascii-printing
[Variable]
char-set:ascii-whitespace
[Variable]
char-set:ascii-blank
[Variable]
char-set:ascii-control
[Variable]
char-set:ascii-punctuation
[Variable]
char-set:ascii-symbol
[Variable]
char-set:ascii-word
[Variable]

These are intersection of char-set:ascii and the corresponding char set without ascii-.
(char-set:ascii-control corresponds to char-set:iso-control).

The \d, \s and \w notation in the char-set literal and regexp literal corresponds to char-set:ascii-digit, char-set:ascii-whitespace, and char-set:ascii-word, respectively (not the Unicode set).

The POSIX character class notation, such as [:alpha:] in char-set literal and regexp literal, refers to these ASCII-only charsets.

Note: We don’t have char-set:ascii-title-case and char-set:ascii-hex-digit. There’s no titlecase letter in ASCII range. And char-set:hex-digit is limited to ASCII by definition.

char-set:Lu
[Variable]
char-set:Ll
[Variable]
char-set:Lt
[Variable]
char-set:Lm
[Variable]
char-set:Lo
[Variable]
char-set:Mn
[Variable]
Each character set contains the corresponding Unicode characters with the given general category; e.g. \texttt{char-set:Lu} contains all characters of the general category \texttt{Lu}.

Each character set contains the Unicode characters with the general category starting with the letter; e.g. \texttt{char-set:L} is union of \texttt{char-set:Lu}, \texttt{char-set:Ll}, \texttt{char-set:Lt}, \texttt{char-set:Lm} and \texttt{char-set:Lo}.

\texttt{char-set:LC} is for cased-letters, the union of \texttt{char-set:Lt}, \texttt{char-set:Ll}, \texttt{char-set:Lu}.

### 6.10.3 Character set operations

See also Section 10.3.6 [R7RS character sets], page 614, for the comprehensive character set operations.

\texttt{char-set? obj} \hspace{1cm} [Function]
\texttt{[R7RS charset]} Returns true if and only if \texttt{obj} is a character set object.

\texttt{char-set-immutable? char-set} \hspace{1cm} [Function]
\texttt{Retruns \#t if char-set is an immutable char-set, \#f if it's a mutable char-set.}

\texttt{char-set-contains? char-set char} \hspace{1cm} [Function]
\texttt{[R7RS charset]} Returns true if and only if a character set object \texttt{char-set} contains a character \texttt{char}.
(char-set-contains? #\[a-z] #\y) ⇒ #t  
(char-set-contains? #\[a-z] #\3) ⇒ #f

(char-set-contains? #\[^ABC] #\A) ⇒ #f  
(char-set-contains? #\[^ABC] #\D) ⇒ #t

char-set char  
A char-set object can be applied to a character, and it works just like (char-set-contains? char-set char).

  (#\[a-z] #\a) ⇒ #t  
  (#\[a-z] #\A) ⇒ #f

(use gauche.collection)

(filter #\[a-z] "CharSet") ⇒ (#\h #\a #\r #\e #\t)

char-set char . . .  
[R7RS charset] Creates a character set that contains char . . . .

(char-set #\a #\b #\c) ⇒ #\[a-c]

char-set-size char-set  
[R7RS charset] Returns a number of characters in the given charset.

gosh> (char-set-size #\[])
0

gosh> (char-set-size #\[:alnum:]]
62

char-set-copy char-set  
[R7RS charset] Copies a character set char-set.

char-set-complement char-set  
[R7RS charset] Returns a complement set of char-set. The former always returns a new set, while the latter may reuse the given charset.

6.11 Strings

<string>  
A string class.

It should be emphasized that Gauche’s internal string object, string body, is immutable. To comply R7RS in which strings are mutable, a Scheme-level string object is an indirect pointer to a string body. Mutating a string means that Gauche creates a new immutable string body that reflects the changes, then swap the pointer in the Scheme-level string object.

This may affect some assumptions on the cost of string operations.

• Copying string is O(1), no matter how long the string is, since the same string body is shared.

• Taking substring usually is also O(1), for the resulting string shares the substring of the original string body. Gauche may copy a part of the string for better memory management, but the visible cost should stay pretty close to O(1). (However, note that accessing to a specific point by index within the original string may cost O(N) because of multibyte string; which is a different story).
Chapter 6: Core library 171

- On the other hand, mutating a string cost \(O(N)\) where \(N\) is the length of string, even for replacing a character.

Gauche does not attempt to make string mutation faster; \((\text{string-set! s k c})\) is exactly as slow as to take two substrings, before and after of \(k\)-th character, and concatenate them with a single-character string in between. So, just avoid string mutations; we believe it’s a better practice. See also Section 6.11.3 [String constructors], page 172.

R7RS string operations are very minimal. Gauche supports some extra built-in operations, and also a rich string library defined in SRFI-13. See Section 11.6 [String library], page 692, for details about SRFI-13.

### 6.11.1 String syntax

" . . . " \[Reader Syntax\]

- [R7RS+] Denotes a literal string. Inside the double quotes, the following backslash escape sequences are recognized.
  - \" [R7RS] Double-quote character
  - \[ R7RS] Backslash character
  - \n [R7RS] Newline character (ASCII 0x0a).
  - \r [R7RS] Return character (ASCII 0x0d).
  - \f [R7RS] Form-feed character (ASCII 0x0c).
  - \t [R7RS] Tab character (ASCII 0x09).
  - \a [R7RS] Alarm character (ASCII 0x07).
  - \b [R7RS] Backspace character (ASCII 0x08).
  - \0 [R7RS] ASCII NUL character (ASCII 0x00).

  \<whitespace>*<newline><whitespace>* [R7RS] Ignored. This can be used to break a long string literal for readability. This escape sequence is introduced in R6RS.

  \xN; [R7RS] A character whose Unicode codepoint is represented by hexadecimal number \(N\), which is any number of hexadecimal digits. (See the compatibility notes below.)

  \uNNNN A character whose UCS2 code is represented by four-digit hexadecimal number NNNN.

  \UNNNNNNNN A character whose UCS4 code is represented by eight-digit hexadecimal number NNNNNNNN.

The following code is an example of backslash-newline escape sequence:

```
(define *message* "\n  This is a long message \n  in a literal string.")
```

*message* ⇒ "This is a long message in a literal string."

Note the whitespace just after ‘message’. Since any whitespaces before ‘in’ is eaten by the reader, you have to put a whitespace between ‘message’ and the following backslash. If you
want to include an actual newline character in a string, and any indentation after it, you can put \"\n\" in the next line like this:

```
(define *message/newline* "\n
This is a long message, \n\nwith a line break."
)
```

Note for the compatibility: We used to recognize a syntax \xNN (two-digit hexadecimal number, without semicolon terminator) as a character in a string; for example, \"\x0d\x0a\" was the same as \"\r\n\". We still support it when we don’t see the terminating semicolon, for the compatibility. There are ambiguous cases: \"\0x0a;\" means \"\n\" in the current syntax, while \"\n;\" in the legacy syntax.

Setting the reader mode to legacy restores the old behavior. Setting the reader mode to warn-legacy makes it work like the default behavior, but prints warning when it finds legacy syntax. See Section 6.21.7.2 [Reader lexical mode], page 272, for the details.

### 6.11.2 String predicates

- **string?** *obj*  
  [Function]  
  [R7RS base] Returns #t if *obj* is a string, #f otherwise.

- **string-immutable?** *obj*  
  [Function]  
  Returns #t if *obj* is an immutable string, #f otherwise

String literals, and the strings returned from certain procedures such as symbol->string are immutable. To ensure you get an immutable string in a program, you can use string-copy-immutable.

- **string-incomplete?** *obj*  
  [Function]  
  Returns #t if *obj* is an incomplete string, #f otherwise

### 6.11.3 String constructors

- **make-string** *k :optional char*  
  [Function]  
  [R7RS base] Returns a string of length *k*. If optional *char* is given, the new string is filled with it. Otherwise, the string is filled with a whitespace. The result string is always complete.

```
(make-string 5 #\x) ⇒ "xxxxx"
```

Note that the algorithm to allocate a string by make-string and then fills it one character at a time is extremely inefficient in Gauche, and should be avoided.

In Gauche, a string is simply a pointer to an immutable string content. If you mutate a string by, e.g. string-set!, Gauche allocates whole new immutable string content, copies the original content with modification, then swap the pointer of the original string. It is no more efficient than making a new copy.

You can use an output string port for a string construction (see Section 6.21.5 [String ports], page 268). Even creating a list of characters and using list->string is faster than using make-string and string-set!.

- **make-byte-string** *k :optional byte*  
  [Function]  
  Creates and returns an incomplete string o size *k*. If *byte* is given, which must be an exact integer, and its lower 8 bits are used to initialize every byte in the created string.

- **string** *char . . .*  
  [R7RS base] Returns a string consisted by *char . . .*
**x->string obj**  
A generic coercion function. Returns a string representation of obj. The default methods are defined as follows: strings are returned as is, numbers are converted by `number->string`, symbols are converted by `symbol->string`, and other objects are converted by `display`.

Other class may provide a method to customize the behavior.

### 6.11.4 String interpolation

The term "string interpolation" is used in various scripting languages such as Perl and Python to refer to the feature to embed expressions in a string literal, which are evaluated and then their results are inserted into the string literal at run time.

Scheme doesn’t define such a feature, but Gauche implements it as a reader macro.

#### string-literal

Evaluates to a string. If `string-literal` contains the character sequence `~expr`, where `expr` is a valid external representation of a Scheme expression, `expr` is evaluated and its result is inserted in the original place (by using `x->string`, see Section 6.11.3 [String constructors], page 172).

The tilde and the following expression must be adjacent (without containing any whitespace characters), or it is not recognized as a special sequence.

To include a tilde itself immediately followed by non-delimiting character, use `~~`.

Other characters in the `string-literal` are copied as is.

If you use a variable as `expr` and need to delimit it from the subsequent string, you can use the symbol escape syntax using ‘\’ character, as shown in the last two examples below.

```scheme
#"This is Gauche, version ~(gauche-version)."
⇒ "This is Gauche, version 0.9.13."

#"Date: ~(sys-strftime "%Y/%m/%d" (sys-localtime (sys-time)))"
⇒ "Date: 2002/02/18"

(let ((a "AAA")
(b "BBB"))

#"xxx ~a ~b zzz")
⇒ "xxx AAA BBB zzz"

#"123~~456~~789"
⇒ "123~456~789"

(let ((n 7)) #"R~|n|RS")
⇒ "R7RS"

(let ((x "bar")) #"foo~|x|.")
⇒ "foobar"
```

In fact, the reader expands this syntax into a macro call, which is then expanded into a call of `string-append` as follows:

```scheme
#"This is Gauche, version ~(gauche-version)."
≡
(string-interpolate* ("This is Gauche, version "
(gauche-version) 
".")))"
;; then, it expands to...

(string-append "This is Gauche, version "
  (x->string (gauche-version))
  ",")

(NB: The exact spec of string-interpolate might change in future, so do not rely on the current behavior.)

Since the #"..." syntax is equivalent to a macro call of string-interpolate, which is provided in the Gauche module, it must be visible from where you use the interpolation syntax. When you write Gauche code, typically you implicitly inherit the Gauche module so you don’t need to worry; however, if you start from R7RS code, make sure you import string-interpolate (by (import (gauche base)), for example) whenever you use string interpolation syntax. Also be careful not to shadow string-interpolate locally.

#\'string-literal  
This is the old style of string-interpolation. It is still recognized, but discouraged for the new code.

Inside string-literal, you can use ,expr (instead of ~expr) to evaluate expr. If comma isn’t immediately followed by a character starting an expression, it loses special meaning.

#\'"This is Gauche, version , (gauche-version)"

Rationale of the syntax: There are wide variation of string interpolation syntax among scripting languages. They are usually linked with other syntax of the language (e.g. prefixing $ to mark evaluating place is in sync with variable reference syntax in some languages).

The old style of string interpolation syntax was taken from quasiquote syntax, because those two are conceptually similar operations (see Section 4.9 [Quasiquotation], page 66). However, since comma character is frequently used in string literals, it was rather awkward.

We decided that tilde is more suitable as the unquote character for the following reasons.

• Traditionally, Lisp’s string formatter format uses ~ to introduce format directives (see Section 6.21.8.4 [Formatting output], page 279). Lispers are used to scan ~’s in a string as variable portions.

• Gauche’s ~ is a universal accessor, and the operator has a nuance of “taking something out of it” (see Section 6.15.2 [Universal accessor], page 219).

• Clojure, a new Lisp dialect, adopted ~ as the unquote character in the quasiquote syntax, instead of commas.

Note that Scheme allows wider range of characters for valid identifier names than usual scripting languages. Consequently, you will almost always need to use ‘!’ delimiters when you interpolate the value of a variable. For example, while you can write "$year/$month/$day $hour:$minutes:$seconds" in Perl, you should write #""|year|/"|month|/"day 
"|hour|:"|minutes|:"seconds". It may be better always to delimit direct variable references in this syntax to avoid confusion.

6.11.5 String cursors

String cursors are opaque objects that point into strings, similar to indexes. Cursors however are more efficient. For example, to get a character with string-ref using an index on a multibyte string, Gauche needs to iterate from the beginning of the string until that position, or O(n).

Using cursors you can access in O(1) (for singlebyte (ASCII) strings or an indexed string, Gauche does it in O(1) even with index. See Section 6.11.6 [String indexing], page 176, for the details of indexed string.)
For a string of length $n$, there can be $n+1$ cursors. The last cursor at the end of the string does not point to any valid character, it’s usually used to determine if nothing is found.

A string cursor is associated with a specific string and should not be used with another string. A string cursor also becomes invalid when the associated string is modified. Accessing an invalid cursor does not always fail though. Running `gosh` with `-fsafe-string-cursors` could help catch these issues, with some performance overhead. See Section 3.1 [Invoking Gosh], page 19.

Most of the time, string cursors aren’t heap-allocated. It is only allocated in heap either (1) when it points at a huge byte index, or (2) when you use `-fsafe-string-cursors` to enable extra run-time check.

The threshold of byte index to cause a string cursor to be heap-allocated is $2^{56}$ on 64bit systems, and $2^{24}$ on 32bit systems, in the current implementation. On 64bit systems you will never hit the threshold practically. On 32bit systems you may, if you have a huge string, but you may want to consider using other data structure rather than keeping such data in one string object.

Most procedures that take indexes in Gauche can also take cursors. Relying on this though is unportable. For example, the `substring` procedure in RnRS standards does not mention anything about cursors even though the Gauche version accepts cursors. For portable programs, you should only use cursors on procedures from `srfi.130` module (see Section 11.28 [Cursor-based string library], page 738).

**<string-cursor>** [Builtin Class]

Represents a cursor. When printed out, you’ll see the byte offset from the beginning of the string, not the character index.

```
(string-index->cursor "" 2)  ⇒ #<string-cursor 6>
```

**string-cursor? obj** [Function]

[SRFI-130] Returns `#t` if `obj` is a string cursor, `#f` otherwise.

**string-cursor-start str** [Function]

[SRFI-130] Returns a cursor pointing to the start of a string `str`. It returns a valid cursor on an empty string too. It’s the same as `string-cursor-end` in that case.

**string-cursor-end str** [Function]

[SRFI-130] Returns a cursor pointing to the end of `str` (the point after the last character.) If `str` is empty, it is the same as `string-cursor-start`. This cursor does not point to any valid character of the string.

**string-cursor-next str cur** [Function]

[SRFI-130] Returns the cursor into `str` following `cur`. `cur` can also be an index. An error is signaled if `cur` points to the end of the string.

**string-cursor-prev str cur** [Function]

[SRFI-130] Returns the cursor into `str` preceding `cur`. `cur` can also be an index. An error is signaled if `cur` points to the beginning of the string.

**string-cursor-forward str cur n** [Function]

[SRFI-130] Returns the cursor into `str` following `cur` by `n` characters. `cur` can also be an index.

**string-cursor-back str cur n** [Function]

[SRFI-130] Returns the cursor into `str` preceding `cur` by `n` characters. `cur` can also be an index.
Chapter 6: Core library

string-index->cursor str index
[Function]
[SRFI-130] Convert an index to a cursor. If index is a cursor it will be returned as-is.

string-cursor->index str cur
[Function]
[SRFI-130] Convert a cursor to an index. If cur is a an index it will be returned as-is.

string-cursor-diff str start end
[Function]
[SRFI-130] Returns the number of characters between start and end. It should be non-negative if start precedes end, non-positive otherwise. start and end also accept index.

string-cursor=? cur1 cur2
[Function]

string-cursor<? cur1 cur2
[Function]

string-cursor<=? cur1 cur2
[Function]

string-cursor>? cur1 cur2
[Function]

string-cursor=>? cur1 cur2
[Function]
[SRFI-130] Compares two cursors or two indexes (but not a cursor and an index) and returns #t or #f accordingly.

6.11.6 String indexing
Since Gauche stores strings in multibyte encoding, random access requires O(N) by default. In most cases, string access is either sequential or search-and-extract pattern, and Gauche provides direct means for these operations, so you don’t need to deal with indexed access. However, there may be a case that you have need more efficient random access string (mostly when porting third-party code, we imagine).

There are a couple of ways to achieve O(1) random access.

First, instead of integer character indexes, you can use string cursors (see Section 6.11.5 [String cursors], page 174). It is defined by srfi.130, and you can use the code that’s using SRFI-130 as is, without worrying about slow access. However, if external interface gives you integer character index, converting index to cursor and vice versa takes O(N) after all.

There’s another way. You can precompute string index, mapping from integer character index to the position in the multibyte string. It costs O(N) of time and space to compute it, but once computed, you have O(1) random access. (We store positions for every K characters, where K is between 16 to 256, so it won’t take up as large storage as the actual string body).

For portability, SRFI-135 Immutable Texts provides O(1) accessible string as “texts”. On Gauche, a text is just an immutable string with index attached.

string-build-index! str
[Function]
Computes and attaches index to a string str, and returns str itself. The operation doesn’t alter the content of str, and you can pass immutable string as well.

If str is a single-byte string (ASCII-only, or incomplete), or a short one (less than 64 octets), no index is attached. It is ok to pass a string which already has an index; then index computation is skipped.

The index is attached to the string’s content. If you alter str by e.g. string-set!, the index is discarded.

string-fast-indexable? str
[Function]
Returns #t if index access of a string str is effectively O(1), that is, str is either a single-byte string, a short string, or a long multibyte string with index computed.
6.11.7 String accessors & modifiers

**string-length** string

[Function]

[R7RS base] Returns a length of (possibly incomplete) string string.

**string-size** string

[Function]

Returns a size of (possibly incomplete) string. A size of string is a number of bytes string occupies on memory. The same string may have different sizes if the native encoding scheme differs.

For incomplete string, its length and its size always match.

**string-ref** cstring k :optional fallback

[Function]

[R7RS+] Returns k-th character of a complete string cstring. It is an error to pass an incomplete string.

By default, an error is signaled if k is out of range (negative, or greater than or equal to the length of cstring). However, if an optional argument fallback is given, it is returned in such case. This is Gauche’s extension.

If cstring is a multibyte string without index attached, this procedure takes O(k) time. See Section 6.11.6 [String indexing], page 176, for ensuring O(1) access.

k can also be a string cursor (also Gauche’s extension). Cursor access is O(1).

**string-byte-ref** string k

[Function]

Returns k-th byte of a (possibly incomplete) string string. Returned value is an integer in the range between 0 and 255. k must be greater than or equal to zero, and less than (string-size string).

**string-set!** string k char

[Function]

[R7RS base] Substitute string’s k-th character by char. k must be greater than or equal to zero, and less than (string-length string). Return value is undefined.

If string is an incomplete string, integer value of the lower 8 bits of char is used to set string’s k-th byte.

See the notes in make-string about performance consideration.

**string-byte-set!** string k byte

[Function]

Substitute string’s k-th byte by integer byte. byte must be in the range between 0 to 255, inclusive. k must be greater than or equal to zero, and less than (string-size string). If string is a complete string, it is turned to incomplete string by this operation. Return value is undefined.

6.11.8 String comparison

**string=?** string1 string2 string3 . . .

[Function]

[R7RS base] Returns #t iff all arguments are strings with the same content.

If any of arguments is incomplete string, it returns #t iff all arguments are incomplete and have exactly the same content. In other words, a complete string and an incomplete string never equal to each other.

**string<?** string1 string2 string3 . . .

[Function]

**string<=?** string1 string2 string3 . . .

[Function]

**string>?** string1 string2 string3 . . .

[Function]

**string=>?** string1 string2 string3 . . .

[Function]

[R7RS base] Compares strings in codepoint order. Returns #t iff all the arguments are ordered.
Comparison between an incomplete string and a complete string, or between two incomplete strings, are done by octet-to-octet comparison. If a complete string and an incomplete string have exactly the same binary representation of the content, a complete string is smaller.

\[
\text{string-ci=} \text{string1 string2 string3} \ldots
\]

\[
\text{string-ci<=} \text{string1 string2 string3} \ldots
\]

\[
\text{string-ci>=} \text{string1 string2 string3} \ldots
\]

Case-insensitive string comparison.

These procedures fold argument character-wise, according to Unicode-defined character-by-character case mapping. See \texttt{char-foldcase} for the details (Section 6.9 \texttt{Characters}, page 160). Character-wise case folding doesn’t handles the case like German eszett:

\[
(\text{string-ci=} "\u00df" "SS") \Rightarrow \#f
\]

R7RS requires \texttt{string-ci*} procedures to use string case folding. Gauche provides R7RS-conformant case insensitive comparison procedures in \texttt{gauche.unicode} (see Section 9.37.3 [Full string case conversion], page 554). If you write in R7RS, importing \texttt{(scheme char)} library, you’ll use \texttt{gauche.unicode}’s \texttt{string-ci*} procedures.

### 6.11.9 String utilities

\textbf{substring \texttt{string start end}}

[R7RS base] Returns a substring of \texttt{string}, starting from \texttt{start}-th character (inclusive) and ending at \texttt{end}-th character (exclusive). The \texttt{start} and \texttt{end} arguments must satisfy \(0 \leq \texttt{start} < \texttt{N}, 0 \leq \texttt{end} \leq \texttt{N}\), and \(\texttt{start} \leq \texttt{end}\), where \(\texttt{N}\) is the length of the string. \texttt{start} and \texttt{end} can also be string cursors, but this is an extension of Gauche.

When \texttt{start} is zero and \texttt{end} is \(\texttt{N}\), this procedure returns a copy of \texttt{string}. (See also \texttt{opt-substring} below, if you don’t want to copy if not necessary.)

Actually, extended \texttt{string-copy} explained below is a superset of \texttt{substring}. This procedure is kept mostly for compatibility of R7RS programs. See also \texttt{subseq} in Section 9.31 [Sequence framework], page 510, for the generic version.

\textbf{opt-substring \texttt{string :optional start end}}

Like \texttt{substring}, returns a part of \texttt{string} between \texttt{start}-th character (inclusive) and \texttt{end}-th character (exclusive). However, if the entire \texttt{string} is used (e.g. \texttt{start} is 0 and \texttt{end} is the length of \texttt{string}, or the arguments are omitted, etc.), \texttt{string} is returned as is, without copying.

This is a typical handling of optional \texttt{start/end} indexes for many string utilities. Note that using \texttt{substring} forces copying the input string even when it’s not necessary.

Besides exact integers, \texttt{#f} or \texttt{#<undefined>} is allowed as \texttt{start} and \texttt{end}, to indicate the argument is missing. In that case, 0 is assumed for \texttt{start}, and the length of \texttt{string} is assumed for \texttt{end}.

\textbf{string-append \texttt{string} \ldots}

[R7RS base] Returns a newly allocated string whose content is concatenation of \texttt{string} \ldots . . .

See also \texttt{string-concatenate} in Section 11.6.9 [SRFI-13 String reverse & append], page 698.

\textbf{string->list \texttt{string :optional start end}}

[Function]

\textbf{list->string \texttt{list}}

[Function]

[SRFI base] Converts a string to a list of characters or vice versa.

You can give an optional start/end indexes to \texttt{string->list}.

For \texttt{list->string}, every elements of \texttt{list} must be a character, or an error is signaled. If you want to build a string out of a mixed list of strings and characters, you may want to use \texttt{tree->string} in Section 12.80 [Lazy text construction], page 1014.
string-copy string :optional start end  
[Function]  
[R7RS base] Returns a copy of string. You can give start and/or end index to extract the part of the original string (it makes string-copy a superset of substring effectively).  
If only start argument is given, a substring beginning from start-th character (inclusive) to the end of string is returned. If both start and end argument are given, a substring from start-th character (inclusive) to end-th character (exclusive) is returned. See substring above for the condition that start and end should satisfy.  
Node: R7RS’s destructive version string-copy! is provided by srfi.13 module (see Section 11.6 [String library], page 692).

string-copy-immutable string :optional start end  
[Function]  
If string is immutable, return it as is. Otherwise, returns an immutable copy of string. It is a dual of string-copy which always returns a mutable copy.  
The optional start and end argument may be a nonnegative integer character index and/or string cursors to restrict the range of string to be copied.

string-fill! string char :optional start end  
[Function]  
[R7RS base] Fills string by char. Optional start and end limits the effective area.

```
(string-fill! "orange" #\X)  
⇒ "XXXXXX"
(string-fill! "orange" #\X 2 4)  
⇒ "orXXge"
```

See the notes in make-string about performance consideration.

string-join strs :optional delim grammar  
[Function]  
[SRFI-13] Concatenate strings in the list strs, with a string delim as ‘glue’.  
The argument grammar may be one of the following symbol to specify how the strings are concatenated.

infix  
Use delim between each string. This mode is default. Note that this mode introduce ambiguity when strs is an empty string or a list with a null string.

```
(string-join '("apple" "mango" "banana")",")  
⇒ "apple, mango, banana"
(string-join '()":"))  
⇒ ""
(string-join '("")":"))  
⇒ ""
```

strict-infix  
Works like infix, but empty list is not allowed to strs, thus avoiding ambiguity.

prefix  
Use delim before each string.

```
(string-join '("usr" "local" "bin")="/"'prefix)  
⇒ "/usr/local/bin"
(string-join '()="/"'prefix)  
⇒ ""
(string-join '("")="/"'prefix)  
⇒ "="/"
```

suffix  
Use delim after each string.

```
(string-join '("a" "b" "c")"&"'suffix)  
⇒ "a&b&c&"
(string-join '()"&"'suffix)  
⇒ "&"
```
⇒ ""
(string-join "("") "&" 'suffix)
⇒ "&"

**string-scan**  
string item :optional return  

[string-scan](#) Scan item (either a string or a character) in string. While **string-scan** finds the leftmost match, **string-scan-right** finds the rightmost match.  

The return argument specifies what value should be returned when item is found in string. It must be one of the following symbols.

- **index** Returns the index in string if item is found, or #f. This is the default behavior.
  
  (string-scan "abracadabra" "ada") ⇒ 5
  
  (string-scan "abracadabra" #\c) ⇒ 4
  
  (string-scan "abracadabra" "aba") ⇒ #f

- **before** Returns a substring of string before item, or #f if item is not found.
  
  (string-scan "abracadabra" "ada" 'before) ⇒ "abrac"
  
  (string-scan "abracadabra" #\c 'before) ⇒ "abra"

- **after** Returns a substring of string after item, or #f if item is not found.
  
  (string-scan "abracadabra" "ada" 'after) ⇒ "bra"
  
  (string-scan "abracadabra" #\c 'after) ⇒ "adabra"

- **before*** Returns a substring of string before item, and the substring after it. If item is not found, returns (values #f #f).
  
  (string-scan "abracadabra" "ada" 'before*)
  ⇒ "abrac" and "adabra"
  
  (string-scan "abracadabra" #\c 'before*)
  ⇒ "abra" and "cadabra"

- **after*** Returns a substring of string up to the end of item, and the rest. If item is not found, returns (values #f #f).
  
  (string-scan "abracadabra" "ada" 'after*)
  ⇒ "abracada" and "bra"
  
  (string-scan "abracadabra" #\c 'after*)
  ⇒ "abrac" and "adabra"

- **both** Returns a substring of string before item and after item. If item is not found, returns (values #f #f).
  
  (string-scan "abracadabra" "ada" 'both)
  ⇒ "abrac" and "bra"
  
  (string-scan "abracadabra" #\c 'both)
  ⇒ "abra" and "adabra"

**string-split**  
string splitter :optional grammar limit start end  

[string-split](#) Splits string by splitter and returns a list of strings. splitter can be a character, a character set, a string, a regexp, or a procedure.

If splitter is a character or a string, it is used as a delimiter. Note that SRFI-152's **string-split** only allows strings for splitter (it also interprets the first optional argument as a grammar; see below for the compatibility note.)  

If splitter is a character set, any consecutive characters that are member of the character set are used as a delimiter.
If a procedure is given to `splitter`, it is called for each character in `string`, and the consecutive characters that caused `splitter` to return a true value are used as a delimiter.

```
(string-split "/aa/bb//cc" "/")  ⇒ ("" "aa" "bb" "" "cc")
(string-split "/aa/bb//cc" "/")  ⇒ ("" "aa" "bb" "" "cc")
(string-split "/aa/bb/cc" "//")  ⇒ ("/aa/bb" "cc")
(string-split "/aa/bb//cc" "/[\]")  ⇒ ("" "aa" "bb" "cc")
(string-split "/aa/bb//cc" "/\+/")  ⇒ ("" "aa" "bb" "cc")
(string-split "/aa/bb//cc" "/\[\])  ⇒ ("" "aa" "bb" "cc")
```

```
(string-split "/aa/bb//cc" char-alphabetic?)  ⇒ ("/" "/" "//" "")
```

;;; some boundary cases
```
(string-split "abc" "/")  ⇒ ("abc")
(string-split "" "/")  ⇒ ("")
```

The `grammar` argument is the same as `string-join` above; it must be one of symbols `infix`, `strict-infix`, `prefix` or `suffix`. When omitted, `infix` is assumed.

```
(string-split "/a/b/c/" "/" 'infix)  ⇒ ("" "a" "b" "c" "")
(string-split "/a/b/c/" "/" 'prefix)  ⇒ ("a" "b" "c" "")
(string-split "/a/b/c/" "/" 'suffix)  ⇒ ("" "a" "b" "c")
```

In general, the following relationship holds:

```
(string-join XS DELIM GRAMMAR)  ⇒ S
(string-split S DELIM GRAMMAR)  ⇒ XS
```

If `limit` is given and not #f, it must be a nonnegative integer and specifies the maximum number of match to the `splitter`. Once the limit is reached, the rest of string is included in the result as is.

```
(string-split "a.b..c" "." 'infix 0)  ⇒ ("a.b..c")
(string-split "a.b..c" "." 'infix 1)  ⇒ ("a" "b..c")
(string-split "a.b..c" "." 'infix 2)  ⇒ ("a" "b" ".c")
```

Compatibility note: The `grammar` argument is added for the consistency of srfis (SRFI-130, SRFI-152, see Section 11.29 [String library (reduced)], page 740). However, for the backward compatibility and the convenience, it also accepts `limit` without `grammar` argument; it is distinguishable since `grammar` is a symbol and `limit` is an integer. For the code that’s compatible to SRFI-152, use the first form that takes `grammar` argument.

```
(string-split "a.b..c" "." 2)  ⇒ ("a" "b" ".c")
```

The `start` and `end` arguments limits input string in the given range before splitting.

See also `string-tokenize` in (see Section 11.6.12 [SRFI-13 Other string operations], page 700).

```
string-map proc str str2 ...
```

[Function]

```
string-map proc str :optional start end
```

[Function]

[R7RS base][SRFI-13] Applies `proc` over each character in the input string, and gathers the characters returned from `proc` into a string and returns it. It is an error if `proc` returns non-character.

Because of historical reasons, this procedure has two interfaces. The first one takes one or more input strings, and `proc` receives as many characters as the number of input strings, each character being taken from each string. Iteration stops on the shortest string. This is defined in R7RS-small, and consistent with `map`, `vector-map`, etc.

The second one takes only one string argument, and optional start/end arguments, which may be nonnegative integer indexes or string cursors to limit the input range of the string. This is defined in SRFI-13, string library.
The order in which proc is applied is not guaranteed to be left to right. You shouldn’t depend on the order.

If proc saves a continuation and it is invoked later, the result already returned from string-map won’t be affected (as specified in R7RS).

(string-map char-upcase "apple") ⇒ "APPLE"
(string-map (^[a b] (if (char>? a b) a b)) "orange" "apple") ⇒ "orpng"
(string-map char-upcase "pineapple" 0 4) ⇒ "PINE"

string-for-each proc str str2 ... [Function]
string-for-each proc str :optional start end [Function]

[R7RS base][SRFI-13] Applies proc over each character in the input string in left-to-right order. The results of proc is discarded.

Because of historical reasons, this procedure has two interfaces, first one defined in R7RS and second one defined in SRFI-13. See string-map above for the explanation.

6.11.10 Incomplete strings

A string can be flagged as "incomplete" if it may contain byte sequences that do not consist of a valid multibyte character in the Gauche’s native encoding.

Incomplete strings may be generated in several circumstances; reading binary data as a string, reading a string data that has been 'chopped' in middle of a multibyte character, or concatenating a string with other incomplete strings, for example.

Incomplete strings should be regarded as an exceptional case. It used to be a way to handle byte strings, but now we have u8vector (see Section 6.13.2 [Uniform vectors], page 198) for that purpose. In fact, we’re planning to remove it in the future releases.

Just in case, if you happen to get an incomplete string, you can convert it to a complete string by string-incomplete->complete.

###"..." [Reader Syntax]

Denotes incomplete string. The same escape sequences as the complete string syntax are recognized.

Rationale of the syntax: ** is used for bit vectors. Since an incomplete strings is really a byte vector, it has similarity.

Note: We used ###"..." for an incomplete string on 0.9.9 and before. It turned out that it couldn’t coexist with bitvectors, for ** is a valid bitvector literal (zero-length vector), and " is a delimiter, so ###"..." can be parsed as a zero-length bitvector followed by a string. From 0.9.10, we changed the incomplete string literal to ###"...". It’s a bit lengthy, but incomplete strings are anomalies and shouldn’t be used often anyway.

For the backward compatibility, ###"..." is still read as an incomplete string literal, unless the reader lexical mode is strict-r7 (see Section 6.21.7.2 [Reader lexical mode], page 272, for the details). If the reader lexical mode is warn-legacy, it is read as an incomplete string, but a warning is issued. If the mode is strict-r7, it is read as a zero-length bitvector followed by a string.

In future releases, ###"..." would be warned by default, and later we’ll gradually move to strict-r7 behavior.

string-incomplete->complete str :optional handling filler [Function]

Reinterpret the content of an incomplete string str and returns a newly created complete string from it. The handling argument specifies how to handle the illegal byte sequences in str.

#f If str contains an illegal byte sequence, give up the conversion and returns #f. This is the default behavior.
Omit any illegal byte sequences.

Replace each byte in illegal byte sequences by a character given in \textit{filler} argument, defaulted to \texttt{?}.

Replace each byte in illegal byte sequences by a sequence of \textit{filler} <\texttt{hexdigit}> <\texttt{hexdigit}>. Besides, the \textit{filler} characters in the original string is replaced with \textit{filler} filler.

If \textit{str} is already a complete string, its copy is returned.

The procedure always returns a complete string, except when the \textit{handling} argument is \texttt{#f} (default) and the input is an incomplete string, in which case \texttt{#f} is returned.

When Gauche's internal encoding is utf-8, the procedure works as follows:

\begin{itemize}
\item \texttt{(string-incomplete->complete #*"_abc")}
\hspace{1em}⇒ "_abc" \hspace{1em}; can be represented as a complete string
\item \texttt{(string-incomplete->complete #*"_ab\x80;c")}
\hspace{1em}⇒ #f \hspace{1em}; can’t be represented as a complete string
\item \texttt{(string-incomplete->complete #*"_ab\x80;c" :omit)}
\hspace{1em}⇒ "_abc" \hspace{1em}; omit the illegal bytes
\item \texttt{(string-incomplete->complete #*"_ab\x80;c" :replace \#\_)}
\hspace{1em}⇒ "_ab_c" \hspace{1em}; replace the illegal bytes
\item \texttt{(string-incomplete->complete #*"_ab\x80;c" :escape \#\_)}
\hspace{1em}⇒ "__ab_80c" \hspace{1em}; escape the illegal bytes and escape char itself
\end{itemize}

\subsection*{6.12 Regular expressions}

Gauche has a built-in regular expression engine which is mostly upper-compatible of POSIX extended regular expression, plus some extensions from Perl 5 regex.

A special syntax is provided for literal regular expressions. Also regular expressions are applicable, that is, it works like procedures that match the given string to itself. Combining with these two features enables writing some string matching idioms compact.

\begin{itemize}
\item \texttt{(find #/pattern/ list-of-strings)}
\hspace{1em}⇒ match object or \texttt{#f}
\end{itemize}

\subsubsection*{6.12.1 Regular expression syntax}

\begin{itemize}
\item \texttt{#/regexp-spec/} \hspace{1em}[Reader Syntax]
\item \texttt{#/regexp-spec/i} \hspace{1em}[Reader Syntax]
\end{itemize}

Denotes literal regular expression object. When read, it becomes an instance of \texttt{<regexp>}. If a letter 'i' is given at the end, the created regexp becomes \textit{case-folding regexp}, i.e. it matches in the case-insensitive way.

The advantage of using this syntax over \texttt{string->regexp} is that the regexp is compiled only once. You can use literal regexp inside loop without worrying about regexp compilation overhead. If you want to construct regexp on-the-fly, however, use \texttt{string->regexp}.

Gauche's built-in regexp syntax follows POSIX extended regular expression, with a bit of extensions taken from Perl. (Scheme Regulare Expression (SRE) is also supported as an alternative syntax; see Section 10.3.19 [R7RS regular expressions], page 640, for the details of SRE.)

\begin{itemize}
\item \texttt{re*} \hspace{1em}Matches zero or more repetition of \textit{re}.
\end{itemize}
**Chapter 6: Core library**

**re+**
Matches one or more repetition of re.

**re?**
Matches zero or one occurrence of re.

**re{n}**
Bounded repetition. re{n} matches exactly n occurrences of re. re{n,m} matches at least n and at most m occurrences of re, where n <= m. In the latter form, either n or m can be omitted; omitted n is assumed as 0, and omitted m is assumed infinity.

**re{n}?**
**re+?**
**re??**
Same as the above repetition construct, but these syntaxes use "non-greedy" or "lazy" match strategy. That is, they try to match the minimum number of occurrences of re first, then retry longer ones only if it fails. In the last form either n or m can be omitted. Compare the following examples:

```latex
(rxmatch-substring (#/<.*>/ "<tag1><tag2><tag3>") 0) ⇒ "<tag1><tag2><tag3>"
(rxmatch-substring (#/<.*?>/ "<tag1><tag2><tag3>") 0) ⇒ "<tag1>"
```

**(re...)**
Clustering with capturing. The regular expression enclosed by parenthesis works as a single re. Besides, the string that matches re ... is saved as a submatch.

**(?:re...)**
Clustering without capturing. re ... works as a single re, but the matched string isn’t saved.

**(?<name>re...)**
Named capture and clustering. Like (re...), but adds the name name to the matched substring. You can refer to the matched substring by both index number and the name.

When the same name appears more than once in a regular expression, it is undefined which matched substring is returned as the submatch of the named capture.

**(?!:re...)**
**(?i:re...)**
**(?!:re...)**
Lexical case sensitivity control. (?!:re...) makes re... matches case-insensitively, while (?i:re...) makes re... matches case-sensitively.

Perl’s regexp allows several more flags to appear between ’?’ and ’!’. Gauche only supports above two, for now.

**pattern1|pattern2|...**
Alternation. Matches either one of patterns, where each pattern is re ... .

**\n**
Backreference. n is an integer. Matches the substring captured by the n-th capturing group. (counting from 1). When capturing groups are nested, groups are counted by their beginnings. If the n-th capturing group is in a repetition and has matched more than once, the last matched substring is used.

**\k<name>**
Named backreference. Matches the substring captured by the capturing group with the name name. If the named capturing group is in a repetition and has matched more than once, the last matched substring is used. If there are more than one capturing group with name, matching will succeed if the input matches either one of the substrings captured by those groups.
. Matches any character (including newline).

[char-set-spec]
Matches any of the character set specified by `char-set-spec`. See Section 6.10 [Character sets], page 165, for the details of `char-set-spec`.

\s, \d, \w
Matches a whitespace character (`char-set:ascii-whitespace, #[\u0009-\u000d]`), a digit character(`char-set:ascii-digit, #[0-9]`), or a word-constituent character (`char-set:ascii-word, #[A-Za-z0-9_]`), respectively. Note that they don’t include characters outside ASCII range.

Can be used both inside and outside of character set.

\S, \D, \W
Matches the complement character set of `\s`, `\d` and `\w`, respectively.

\^, \$
Beginning and end of string assertion, when appears at the beginning or end of the pattern, or optionally, beginning and end of line in multi-line mode.

These characters loses special meanings and matches the characters themselves if they appear in the position other than the beginning of the pattern (for `\^`) or the end (for `$`). For the sake of recognizing those characters, lookahead/lookbehind assertions (`(?=...), (?!...), (?<=...), (?<!...)`) and atomic clustering (`(?>...)`) are treated as if they are a whole pattern. That is, `\^` at the beginning of those groupings are beginning-of-string assertion no matter where these group appear in the containing regexp. So as `$` at the end of these groupings.

\b, \B
Word boundary and non word boundary assertion, respectively. That is, `\b` matches an empty string between word-constituent character and non-word-constituent character, and `\B` matches an empty string elsewhere.

\;\n\"\n\#
These are the same as `;`, `"`, and `#`, respectively, and can be used to avoid confusing Emacs or other syntax-aware editors that are not familiar with Gauche’s extension.

(?=pattern)
(?!pattern)
Positive/negative lookahead assertion. Match succeeds if `pattern` matches (or does not match) the input string from the current position, but this doesn’t move the current position itself, so that the following regular expression is applied again from the current position.

For example, the following expression matches strings that might be a phone number, except the numbers in Japan (i.e. ones that begin with "81").

`\+(?!81)\d{9,}`

(?<=pattern)
(?<!pattern)
Positive/negative lookbehind assertion. If the input string immediately before the current input position matches `pattern`, this pattern succeeds or fails, respectively. Like lookahead assertion, the input position isn’t changed.

Internally, this match is tried by reversing `pattern` and applies it to the backward of input character sequence. So you can write any regexp in `pattern`, but if the submatches depend on the matching order, you may get different submatches from when you match `pattern` from left to right.
Atomic clustering. Once pattern matches, the match is fixed; even if the following pattern fails, the engine won’t backtrack to try the alternative match in pattern.

They are the same as (?re*), (?re+), (?re?), respectively.

Conditional matching. If test-pattern counts true, then-pattern is tried; otherwise else-pattern is tried when provided.

test-pattern can be either one of the following:

(integer) Backreference. If integer-th capturing group has a match, this test counts true.

(?=pattern)  Positive/negative lookahead assertion. It tries pattern from the current input position without consuming input, and if the match succeeds or fails, respectively, this test counts true.

(?<=pattern)  Positive/negative lookbehind assertion. It tries pattern backward from the left size of the current input position, and if the match succeeds or fails, respectively, this test counts true.

6.12.2 Using regular expressions

Regexp object and rxmatch object

Regular expression object. You can construct a regexp object from a string by string->regexp or sre->regexp at run time. Gauche also has a special syntax to denote regexp literals, which construct regexp object at loading time.

Gauche’s regexp engine is fully aware of multibyte characters.

Regexp match object. A regexp matcher rxmatch returns this object if match. This object contains all the information about the match, including submatches.

The advantage of using match object, rather than substrings or list of indices, is efficiency. The regmatch object keeps internal state of match, and computes indices and/or substrings only when requested. This is particularly effective for multibyte strings, for index access is slow on them.

Takes string as a regexp specification, and constructs an instance of regexp object.

If a true value is given to the keyword argument case-fold, the created regexp object becomes case-folding regexp. (See the above explanation about case-folding regexp).

If a true value is given to the keyword argument multi-line, ^ and $ will assert the beginning and end of line in addition to beginning and end of string. Popular line terminators (LF only, CRLF and CR only) are recognized.
Chapter 6: Core library

sre->regexp sre :key multi-line
Takes a scheme regexp sre and returns a <regexp> object. The zero-th group is always captured.
If a false value is given to the keyword argument multi-line, which is the default, bol and eol behave like bos and eos (i.e. only match at the beginning or end of string).

regexp? obj
Returns true iff obj is a regexp object.

regexp->string regexp
Returns a source string describing the regexp regexp. The returned string is immutable.

regexp->sre regexp
Returns a scheme regexp (SRE) describing the regexp regexp. See Section 10.3.19 [R7RS regular expressions], page 640, for the details of SRE.

regexp-num-groups regexp
regexp-named-groups regexp
Queries the number of capturing groups, and an alist of named capturing groups, in the given regexp, respectively.
The number of capturing groups corresponds to the number of matches returned by rxmatch-num-matches. Note that the entire regexp forms a group, so the number is always positive.
The alist returned from regexp-named-groups has the group name (symbol) in car, and its subgroup number in cdr. Note that the order of groups in the alist isn’t fixed.

Trying a match

rxmatch regexp string :optional start end
Regexp is a regular expression object. A string string is matched by regexp. If it matches, the function returns a <regmatch> object. Otherwise it returns #f.
If start and/or end are given, only the substring between start (inclusive) and end (exclusive) is searched.
This is called match, regexp-search or string-match in some other Scheme implementations.
Internally, Gauche uses backtracking for regexp match. When regexp has multiple match possibilities, Gauche saves an intermediate result in a stack and try one choice, and if it fails try another. Depending on regexp, the saved results may grow linear to the input. Gauche allocates a fixed amount of memory for that, and if there are too many saved results, you’ll get the following error:

ERROR: Ran out of stack during matching regexp #/.../. Too many retries?
If you get this error, consider using hybrid parsing approach. Our regexp engine isn’t made to do everything-in-one-shot parsing; in most cases, the effect of complex regexp can be achieved better with more powerful grammar than regular grammar.
To apply the match repeatedly on the input string, or to match from the input stream (such as the data from the port), you may want to check grxmatch in gauche.generator (see Section 9.12.2 [Generator operations], page 440).
**regexp string**

A regular expression object can be applied directly to the string. This works the same as `(rxmatch regexp string)`, but allows shorter notation. See Section 6.15.6 [Applicable objects], page 225, for generic mechanism used to implement this.

### Accessing the match result

- **rxmatch-start** `match :optional (i 0)`  
  [Function]
- **rxmatch-end** `match :optional (i 0)`  
  [Function]
- **rxmatch-substring** `match :optional (i 0)`  
  [Function]

*Match* is a match object returned by `rxmatch`. If `i` equals to zero, the functions return start, end or the substring of entire match, respectively. With positive integer `I`, it returns those of `I`-th submatches. It is an error to pass other values to `I`.

It is allowed to pass `#f` to `match` for convenience. The functions return `#f` in such case.

These functions correspond to scsh’s `match:start`, `match:end` and `match:substring`.

- **rxmatch-after** `match :optional (i 0)`  
  [Function]
- **rxmatch-before** `match :optional (i 0)`  
  [Function]

Returns substring of the input string after or before `match`. If optional argument is given, the `i`-th submatch is used (0-th submatch is the entire match).

```
(define match (rxmatch #/(/\d+)\.(\d+)/ "pi=3.14..."))
```

```
(rxmatch-after match)  ⇒  "..."
(rxmatch-after match 1)  ⇒  ".14..."
```

```
(rxmatch-before match)  ⇒  "pi="
(rxmatch-before match 2)  ⇒  "pi=3."
```

- **rxmatch-substrings** `match :optional start end`  
  [Function]
- **rxmatch-positions** `match :optional start end`  
  [Function]

Retrieves multiple submatches (again, 0-th match is the entire match), in substrings and in a cons of start and end position, respectively.

```
(rxmatch-substrings (#/(/\d+)\.(/\d+)/ "12:34:56"))
⇒  (("12:34:56" "12" "34" "56")
```

```
(rxmatch-positions (#/(/\d+)\.(/\d+)/ "12:34:56"))
⇒  ((0 . 8) (0 . 2) (3 . 5) (6 . 8))
```

For the convenience, you can pass `#f` to `match`; those procedures returns `()` in that case.

The optional `start` and `end` arguments specify the range of submatch index. If omitted, `start` defaults to 0 and `end` defaults to `(rxmatch-num-matches match)`. For example, if you don’t need the whole match, you can give 1 to `start` as follows:

```
(rxmatch-substrings (#/(/\d+)\.(/\d+)/ "12:34:56") 1)
⇒  (("12" "34" "56")
```

- **rxmatch->string** `regexp string :optional selector ...`  
  [Function]

A convenience procedure to match a string to the given regexp, then returns the matched substring, or `#f` if it doesn’t match.

If no `selector` is given, it is the same as this:

```
(rxmatch-substring (rxmatch regexp string))
```

If an integer is given as a selector, it returns the substring of the numbered submatch.
If a symbol after or before is given, it returns the substring after or before the match. You can give these symbols and an integer to extract a substring before or after the numbered submatch.

```scheme
(gosh> (rxmatch->string #\d+/ "foo314bar")
"314"
(gosh> (rxmatch->string #/(\w+@([\w.]+)/ "foo@example.com" 2)
"example.com"
(gosh> (rxmatch->string #/(\w+@([\w.]+)/ "foo@example.com" 'before 2)
"foo"
```

**regmatch**:optional index

**regmatch** 'before :optional index

**regmatch** 'after :optional index

A regmatch object can be applied directly to the integer index, or a symbol before or after. They works the same as (rxmatch-substring regmatch index), (rxmatch-before regmatch), and (rxmatch-after regmatch), respectively. This allows shorter notation. See Section 6.15.6 [Applicable objects], page 225, for generic mechanism used to implement this.

```scheme
(define match (#/(\d+)/.(\d+)/ "pi=3.14..."))

(match) ⇒ "3.14"
(match 1) ⇒ "3"
(match 2) ⇒ "14"

(match 'after) ⇒ "..."
(match 'after 1) ⇒ ".14..."

(match 'before) ⇒ "pi="
(match 'before 2) ⇒ "pi=3."

(define match (#/(?<integer>\d+)/.(?<fraction>\d+)/ "pi=3.14..."))

(match 1) ⇒ "3"
(match 2) ⇒ "14"

(match 'integer) ⇒ "3"
(match 'fraction) ⇒ "14"

(match 'after 'integer) ⇒ ".14..."
(match 'before 'fraction) ⇒ "pi=3."
```

**rxmatch-num-matches**:match

**rxmatch-named-groups**:match

Returns the number of matches, and an alist of named groups and whose indices, in *match*. This corresponds regexp-num-groups and regexp-named-groups on a regular expression that has been used to generate *match*. These procedures are useful to inspect *match* object without having the original regexp object.

The number of matches includes the "whole match", so it is always a positive integer for a <regmatch> object. The number also includes the submatches that don’t have value (see the examples below). The result of rxmatch-named-matches also includes all the named groups in the original regexp, not only the matched ones.

For the convenience, rxmatch-num-matches returns 0 and rxmatch-named-groups returns () if *match* is #f.
(rxmatch-num-matches (rxmatch #/abc/ "abc")) ⇒ 1
(rxmatch-num-matches (rxmatch #/(a(.))|(b(.))/ "ba")) ⇒ 5
(rxmatch-num-matches #f) ⇒ 0

(rxmatch-named-groups
(rxmatch #/(?<h>\d\d):(?<m>\d\d):(?<s>\d\d)?/ "12:34")
⇒ ((s . 4) (m . 2) (h . 1))

Convenience utilities

regexp-replace  regexp string substitution  [Function]
regexp-replace-all  regexp string substitution  [Function]

Replaces the part of string that matched to regexp for substitution. regexp-replace just replaces the first match of regexp, while regexp-replace-all repeats the replacing throughout entire string.

substitution may be a string or a procedure. If it is a string, it can contain references to the submatches by digits preceded by a backslash (e.g. \2) or the named submatch reference (e.g. \k<name>). \0 refers to the entire match. Note that you need two backslashes to include backslash character in the literal string; if you want to include a backslash character itself in the substitution, you need four backslashes.

(regexp-replace #/def|DEF/ "abcdefghi" "...")
⇒ "abc...ghi"
(regexp-replace #/def|DEF/ "abcdefghi" "\|\0\")
⇒ "abc\|def\|ghi"
(regexp-replace #/def|DEF/ "abcdefghi" "\|\|\|\0\")
⇒ "abc\|\|\|\0\|ghi"
(regexp-replace #/c(.*)g/ "abcdefghi" "\|\1\")
⇒ "ab\|def\|hi"
(regexp-replace #/c(?<match>.*).*/g/ "abcdefghi" "\|\k<match>|\")
⇒ "ab\|def\|hi"

If substitution is a procedure, for every match in string it is called with one argument, regexp-match object. The returned value from the procedure is inserted to the output string using display.

(regexp-replace #/c(.*)g/ "abcdefghi"
(lambda (m)
  (list->string
   (reverse
    (string->list (rxmatch-substring m 1))))))
⇒ "abfedhi"

Note: regexp-replace-all applies itself recursively to the remaining of the string after match. So the beginning of string assertion in regexp doesn’t only mean the beginning of input string.

Note: If you want to operate on multiple matches in the string instead of replacing it, you can use lrxmatch in gauche.lazy module or grxmatch in gauche.generator module. Both can match a regexp repeatedly and lazily to the given string, and lrxmatch returns a lazy sequence of regmatches, while grxmatch returns a generator that yields regmatches.

(map rxmatch-substring (lrxmatch #/\w+/ "a quick brown fox!?"))
⇒ ("a" "quick" "brown" "fox")
**regexp-replace** string rx1 sub1 rx2 sub2 . . .  
**regexp-replace-all** string rx1 sub1 rx2 sub2 . . .  

First applies **regexp-replace** or **regexp-replace-all** to *string* with a regular expression *rx1* substituting for *sub1*, then applies the function on the result string with a regular expression *rx2* substituting for *sub2*, and so on. These functions are handy when you want to apply multiple substitutions sequentially on a string.

**regexp-quote string**  
Returns a string with the characters that are special to regexp escaped.

```
(regexp-quote "[2002/10/12] touched foo.h and *.c")
⇒ "\[2002/10/12\] touched foo\.h and \*\.c"
```

In the following macros, *match-expr* is an expression which produces a match object or #f. Typically it is a call of **rxmatch**, but it can be any expression.

**rxmatch-let** match-expr (var . . .) form . . .  
Evaluates *match-expr*, and if matched, binds *var* . . . to the matched strings, then evaluates *form*. The first *var* receives the entire match, and subsequent variables receive submatches. If the number of submatches are smaller than the number of variables to receive them, the rest of variables will get #f.

It is possible to put #f in variable position, which says you don’t care that match.

```
(rxmatch-let (rxmatch #/\d+:\d+/) "Jan 1 11:22:33")
   (time)
   (format #f "time is ~a" time)
   "unknown time")
⇒ "time is 11:22"
```

This macro corresponds to scsh’s **let-match**.

**rxmatch-if** match-expr (var . . .) then-form else-form  
Evaluates *match-expr*, and if matched, binds *var* . . . to the matched strings and evaluates *then-form*. Otherwise evaluates *else-form*. The rule of binding vars is the same as **rxmatch-let**.

```
(rxmatch-if (rxmatch #/\d+:\d+/) "Jan 1 11:22:33")
   (time)
   (format #f "time is ~a" time)
   "unknown time")
⇒ "time is 11:22"
```

This macro corresponds to scsh’s **if-match**.
rxmatch-cond clause . . .

Evaluate condition in clauses one by one. If a condition of a clause satisfies, rest portion of the clause is evaluated and becomes the result of \texttt{rxmatch-cond}. Clause may be one of the following pattern.

\[
\text{(match-expr (var ...)} \text{ form ...)}
\]

Evaluate \texttt{match-expr}, which may return a regexp match object or \texttt{#f}. If it returns a match object, the matches are bound to \texttt{vars}, like \texttt{rxmatch-let}, and \texttt{forms} are evaluated.

\[
\text{(test expr form ...)}
\]

Evaluates \texttt{expr}. If it yields true, evaluates \texttt{forms}.

\[
\text{(test expr => proc)}
\]

Evaluates \texttt{expr} and if it is true, calls \texttt{proc} with the result of \texttt{expr} as the only argument.

\[
\text{(else form ...)}
\]

If this clause exists, it must be the last clause. If other clauses fail, \texttt{forms} are evaluated.

If no \texttt{else} clause exists, and all the other clause fail, an undefined value is returned.

\[
\text{;; parses several possible date format}
\]

\[
\text{(define (parse-date str)} \text{)}
\]

\[
\text{(rxmatch-cond)}
\]

\[
\text{(rxmatch #/\(^{\text{100000}}\)/\(^{\text{100000}}\)/\(^{\text{100000}}\)/ str)}
\]

\[
\text{(map string->number (list yyyy mm dd))}
\]

\[
\text{(rxmatch #/\(^{\text{100000}}\)/\(^{\text{100000}}\)/\(^{\text{100000}}\)/ str)}
\]

\[
\text{(map string->number (list yyyy mm dd))}
\]

\[
\text{(rxmatch #/\(^{\text{100000}}\)/\(^{\text{100000}}\)/\(^{\text{100000}}\)/ str)}
\]

\[
\text{(errorf "ambiguous: ~s" str))}
\]

\[
\text{(else (errorf "bogus: ~s" strㄘ))}
\]

\[
\text{(parse-date "2001/2/3")} \Rightarrow \text{(2001 2 3)}
\]

\[
\text{(parse-date "12/25/1999")} \Rightarrow \text{(1999 12 25)}
\]

This macro corresponds to scsh’s \texttt{match-cond}.

rxmatch-case string-expr clause . . .

String-expr is evaluated, and clauses are interpreted one by one. A clause may be one of the following pattern.

\[
\text{(re (var ...)} \text{ form ...)}
\]

\textit{Re} must be a literal regexp object (see Section 6.12 [Regular expressions], page 183). If the result of \texttt{string-expr} matches \texttt{re}, the match result is bound to \texttt{vars} and \texttt{forms} are evaluated, and \texttt{rxmatch-case} returns the result of the last form.

If \texttt{re} doesn’t match the result of \texttt{string-expr}, \texttt{string-expr} yields non-string value, the interpretation proceeds to the next clause.

\[
\text{(test proc form ...)}
\]

A procedure \texttt{proc} is applied on the result of \texttt{string-expr}. If it yields true value, \texttt{forms} are evaluated, and \texttt{rxmatch-case} returns the result of the last \texttt{form}.

If \texttt{proc} yields \texttt{#f}, the interpretation proceeds to the next clause.
(test proc => proc2)
A procedure proc is applied on the result of string-expr. If it yields true
value, proc2 is applied on the result, and its result is returned as the result
of rxmatch-case.
If proc yields #f, the interpretation proceeds to the next clause.

(else form . . )
This form must appear at the end of clauses, if any. If other clauses fail, forms are
evaluated, and the result of the last form becomes the result of rxmatch-case.

(else => proc)
This form must appear at the end of clauses, if any. If other clauses fail, proc is
evaluated, which should yield a procedure taking one argument. The value of
string-expr is passed to proc, and its return values become the return values of
rxmatch-case. rx

If no else clause exists, and all other clause fail, an undefined value is returned.

The parse-date example above becomes simpler if you use rxmatch-case

(define (parse-date2 str)
  (rxmatch-case str
    (test (lambda (s) (not (string? s))) #f)
    (#/^\d\d?/\/(\d\d?)\/(\d\d\d\d)$/ (if mm dd yyyy)
      (map string->number (list yyyy mm dd)))
    (#/^\d\d\d\d/\/(\d\d?)\/(\d\d?)$/ (if yyyy mm dd)
      (map string->number (list yyyy mm dd)))
    (#/^\d+//\d+$/
      (errorf "ambiguous: ^s" str))
    (else (errorf "bogus: ^s" str))))

6.12.3 Inspecting and assembling regular expressions

When Gauche reads a string representation of regexp, first it parses the string and construct
an abstract syntax tree (AST), performs some optimizations on it, then compiles it into an
instruction sequence to be executed by the regexp engine.

The following procedures expose this process to user programs. It may be easier for programs
to manipulate an AST than a string representation.

regexp-parse string :key case-fold multi-line [Function]
Parses a string specification of regexp in string and returns its AST, represented in S-
expression. See below for the spec of AST.
When a true value is given to the keyword argument case-fold, returned AST will match
case-insensitively. (Case insensitive regexp is handled in parser level, not by the engine).

regexp-parse-sre sre [Function]
Parses sre as a Scheme Regular Expression (SRE) as described in SRFI-115 and returns its
AST. See Section 10.3.19 [R7RS regular expressions], page 640, see Section 10.3.19 [R7RS
regular expressions], page 640.

regexp-optimize ast [Function]
Performs some rudimental optimization on the regexp AST, returning regexp AST.
Currently it only optimizes some trivial cases. The plan is to make it cleverer in future.

regexp-compile ast :key multi-line [Function]
Takes a regexp ast and returns a regexp object. Currently the outermost form of ast must
be the zero-th capturing group. (That is, ast should have the form (0 #f x . . .).) The outer
grouping is always added by regexp-parse to capture the entire regexp.
Note: The function does some basic check to see the given AST is valid, but it may not reject invalid ASTs. In such case, the returned regexp object doesn’t work properly. It is caller’s responsibility to provide a properly constructed AST. (Even if it rejects an AST, error messages are often incomprehensible. So, don’t use this procedure as a AST validness checker.)

**regexp-ast** *regexp*  
Returns AST used for the regexp object *regexp*.

**regexp-unparse** *ast :key* (**on-error** *error*)  
From the regexp’s *ast*, reconstruct the string representation of the regexp. The keyword argument **on-error** can be a keyword :**error** (default) or #f. If it’s the former, an error is signaled when *ast* isn’t valid regexp AST. If it’s the latter, **regexp-unparse** just returns #f.

This is the structure of AST. Note that this is originally developed only for internal use, and not very convenient to manipulate from the code (e.g. if you insert or delete a subtree, you have to renumber capturing groups to make them consistent.)

```
<ast> : <clause> ; special clause
  | <item> ; matches <item>

<item> : <char> ; matches char
  | <char-set> ; matches char set
  | (comp . <char-set>) ; matches complement of char set
  | any ; matches any char
  | bos | eos ; beginning/end of string assertion
  | bol | eol ; beginning/end of line assertion
  | bow | eow | wb | nwb ; word-boundary/negative word boundary assertion
  | bog | eog ; beginning/end of grapheme assertion

<clause> : (seq <ast> ...); sequence
  | (seq-uncase <ast> ...); sequence (case insensitive match)
  | (seq-case <ast> ...); sequence (case sensitive match)
  | (alt <ast> ...); alternative
  | (rep <m> <n> <ast> ...); repetition at least <m> up to <n> (greedy)
    ; <n> may be ‘#f’
  | (rep-min <m> <n> <ast> ...); repetition at least <m> up to <n> (lazy)
    ; <n> may be ‘#f’
  | (rep-while <m> <n> <ast> ...); like rep, but no backtrack
  | ((<integer> <symbol> <ast> ...)); capturing group. <symbol> may be #f.
  | (cpat <condition> (<ast> ...)); positive lookahead assertion
  | (assert . <asst>) ; negative lookahead assertion

<condition> : <integer> ; (?(1)yes|no) style conditional expression
  | (assert . <asst>) ; (?(>=condition)...) or (?(<?=condition)...) 
  | (nassert . <asst>) ; (?(?<condition)...) or (?(<!condition)...) 
```

### 6.13 Vector family

Vectors are fixed-size, O(1) accessible sequence of values. Scheme has traditionally offered a vector of arbitrary objects, which is described in Section 6.13.1 [Vectors], page 195.
In R7RS-large, there’re also homogeneous numeric vectors (u vectors), which can contain fixed range of numeric objects efficiently. We explain them in Section 6.13.2 [Uniform vectors], page 198.

Gauche also supports bitvectors, which can contain sequence of bits. See Section 6.13.3 [Bitvectors], page 203, for the details.

Finally, weak vectors are a vector of arbitrary objects using weak pointers. See Section 6.13.4 [Weak vectors], page 205.

**6.13.1 Vectors**

A vector is a simple 1-dimensional array of Scheme objects. You can access its element by index in constant time. Once created, a vector can’t be resized.

Class `<vector>` inherits `<sequence>` and you can use various generic functions such as `map` and `fold` on it. See Section 9.6 [Collection framework], page 404, and See Section 9.31 [Sequence framework], page 510.

If you keep only a homogeneous numeric type, you may be able to use uniform vectors (see Section 6.13.2 [Uniform vectors], page 198).

R7RS defines bytevectors; in Gauche, they’re just `u8vector`s in `gauche.uvector` module (r7rs modules defines aliases. see Section 10.2.2 [R7RS base library], page 585).

See Section 10.3.2 [R7RS vectors], page 597, for additional operations on vectors.

---

**Function** `vector? obj`  
[R7RS base] Returns `#t` if `obj` is a vector, `#f` otherwise.

**Function** `make-vector k :optional fill`  
[R7RS base] Creates and returns a vector with length `k`. If optional argument `fill` is given, each element of the vector is initialized by it. Otherwise, the initial value of each element is undefined.

**Function** `vector obj . . .`  
[R7RS base] Creates a vector whose elements are `obj . . .`.

**Function** `vector-tabulate len proc`  
Creates a vector of length `len`, initializing `i`-th element of which by `(proc i)` for all `i` between 0 and `len`.

```
(vector-tabulate 5 (^x (* x x)))
⇒ #(0 1 4 9 16)
```

**Function** `vector-length vector`  
[R7RS base] Returns the length of a vector `vector`.

With `gauche.collection` module, you can also use a method `size-of`.

**Function** `vector-ref vector k :optional fallback`  
[R7RS+*] Returns `k`-th element of vector `vector`.

By default, `vector-ref` signals an error if `k` is negative, or greater than or equal to the length of `vector`. However, if an optional argument `fallback` is given, it is returned for such case. This is an extension of Gauche.

With `gauche.sequence` module, you can also use a method `ref`.

**Function** `vector-set! vector k obj`  
[R7RS base] Sets `k`-th element of the vector `vector` to `obj`. It is an error if `k` is negative or greater than or equal to the length of `vector`.

With `gauche.sequence` module, you can also use a setter method of `ref`. 
vector->list vector :optional start end  
list->vector list :optional start end

[R7RS+] Converts a vector to a list, or vice versa.

The optional start and end arguments limit the range of the source. (R7RS don’t define start and end arguments for list->vector.)

\[
\begin{align*}
(\text{vector->list '}(1 2 3 4 5)) & \Rightarrow (1 2 3 4 5) \\
(\text{list->vector '}(1 2 3 4 5)) & \Rightarrow '(1 2 3 4 5) \\
(\text{vector->list '}(1 2 3 4 5) 2 4) & \Rightarrow (3 4) \\
(\text{list->vector (circular-list 'a 'b 'c) 1 6}) & \Rightarrow '(b c a b c)
\end{align*}
\]

With gauche.collection module, you can use (coerce-to <list> vector) and (coerce-to <vector> list) as well.

reverse-list->vector list :optional start end

[R7RS vector] Without optional arguments, it returns the same thing as (list->vector (reverse list)), but does not allocate the intermediate list. The optional start and end argument limits the range of the input list.

\[
(\text{reverse-list->vector '(a b c d e f g) 1 5}) \Rightarrow '(e d c b)
\]

vector->string vector :optional start end  
string->vector string :optional start end

[R7RS base] Converts a vector of characters to a string, or vice versa. It is an error to pass a vector that contains other than characters to vector->string.

The optional start and end arguments limit the range of the source.

\[
\begin{align*}
(\text{vector->string '}(\#\text{\(a b c d e\)}) & \Rightarrow "abcde" \\
(\text{string->vector "abcde"}) & \Rightarrow '\#\text{\(a b c d e\)} \\
(\text{vector->string '}(\#\text{\(a b c d e\)}) 2 4) & \Rightarrow "cd"
\end{align*}
\]

With gauche.collection module, you can use (coerce-to <string> vector) and (coerce-to <vector> string) as well.

vector-fill! vector fill :optional start end

[R7RS base] Sets all elements in a vector vector to fill.

Optional start and end limits the range of effect between start-th index (inclusive) to end-th index (exclusive). Start defaults to zero, and end defaults to the length of vector.

vector-copy vector :optional start end fill

[R7RS base] Copies a vector vector. Optional start and end arguments can be used to limit the range of vector to be copied. If the range specified by start and end falls outside of the original vector, the fill value is used to fill the result vector.

\[
\begin{align*}
(\text{vector-copy '}(1 2 3 4 5)) & \Rightarrow '(1 2 3 4 5) \\
(\text{vector-copy '}(1 2 3 4 5) 2 4) & \Rightarrow '(3 4) \\
(\text{vector-copy '}(1 2 3 4 5) 3 7 #f) & \Rightarrow '(4 5 #f #f)
\end{align*}
\]

vector-copy! target tstart source :optional sstart send

[R7RS base] Copies the content of source vector into the target vector starting from tstart in the target. The target vector must be mutable. Optional sstart and send limits the range of source vector.

\[
(rlet1 v (vector 'a 'b 'c 'd 'e) \\
(\text{vector-copy! v 2 '}(1 2))) \Rightarrow '(a b 1 2 e)\]
(rlet1 v (vector ’a ’b ’c ’d ’e)
   (vector-copy! v 2 ’#(1 2 3 4) 1 3))
⇒ #((a b) 2 3 e)

An error is raised if the portion to be copied is greater than the room in the target (that is, between \texttt{tstart} to the end).

It is ok to pass the same vector to \texttt{target} and \texttt{source}; it always works even if the regions of source and destination are overlapping.

\textbf{vector-append} \texttt{vec} \ldots

[Function]

\begin{itemize}
\item \texttt{[R7RS base]} Returns a newly allocated vector whose contents are concatenation of elements of \texttt{vec} in order.
\end{itemize}

\begin{itemize}
\item \texttt{[(vector-append ’#(1 2 3) ’#(a b)) ⇒ #(1 2 3 a b)}
\item \texttt{(vector-append) ⇒ #()}
\end{itemize}

\textbf{vector-map} \texttt{proc vec1 vec2} \ldots

[Function]

\begin{itemize}
\item \texttt{[R7RS base]} Returns a new vector, \(i\)-th of which is calculated by applying \texttt{proc} on the list of each \(i\)-th element of \texttt{vec1 vec2} \ldots. The length of the result vector is the same as the shortest vector of the arguments.
\end{itemize}

\begin{itemize}
\item \texttt{[(vector-map + ’#(1 2 3) ’#(4 5 6 7)) ⇒ #((5 7 9))}
\end{itemize}

The actual order \texttt{proc} is called is undefined, and may change in the future versions, so \texttt{proc} shouldn’t use side effects affected by the order.

Note: If you use \texttt{gauche.collection}, you can get the same function by \texttt{(map-to <vector> proc vec1 vec2 \ldots)}.

\textbf{vector-map-with-index} \texttt{proc vec1 vec2} \ldots

[Function]

\begin{itemize}
\item Like \texttt{vector-map}, but \texttt{proc} receives the current index as the first argument.
\end{itemize}

\begin{itemize}
\item \texttt{[(vector-map-with-index list ’#(a b c d e) ’#(A B C)) ⇒ #((0 a A) (1 b B) (2 c C))}
\end{itemize}

This is what SRFI-43 calls \texttt{vector-map}. See Section 11.12 [Vector library (Legacy)], page 717.

Note: If you use \texttt{gauche.collection}, you can get the same function by \texttt{(map-to-with-index <vector> proc vec1 vec2 \ldots)}.

\textbf{vector-map!} \texttt{proc vec1 vec2} \ldots

[Function]

\begin{itemize}
\item \texttt{[R7RS vector]} For each index \(i\), calls \texttt{proc} with \(i\)-th index of \texttt{vec1 vec2} \ldots, and set the result back to \texttt{vec1}. The value is calculated up to the minimum length of input vectors.
\end{itemize}

\begin{itemize}
\item \texttt{(rlet1 v (vector 1 2 3)
   (vector-map! ($ + 1 $) v)) ⇒ #((2 3 4))}
\item \texttt{(rlet1 v (vector 1 2 3 4)
   (vector-map! + v ’#(10 20))) ⇒ #((11 22 3 4))}
\end{itemize}

\textbf{vector-map-with-index!} \texttt{proc vec1 vec2} \ldots

[Function]

\begin{itemize}
\item Like \texttt{vector-map!}, but \texttt{proc} receives the current index as the first argument. This is equivalent to SRFI-43’s \texttt{vector-map!} (see Section 11.12 [Vector library (Legacy)], page 717).
\end{itemize}

\begin{itemize}
\item \texttt{(rlet1 v (vector ’a ’b ’c)
   (vector-map-with-index! list v)) ⇒ #((0 a) (1 b) (2 c))}
\end{itemize}
vector-for-each proc vec1 vec2 . . .  [Function]
[R7RS base] For all \( i \) below the minimum length of input vectors, calls \( proc \) with \( i \)-th elements of \( vec1 vec2 . . . \), in increasing order of \( i \).

\[
\text{(vector-for-each print '('#(a b c))}
\Rightarrow \text{prints a, b and c.}
\]

vector-for-each-with-index proc vec1 vec2 . . .  [Function]
Like vector-for-each, but \( proc \) receives the current index in the first argument.

This is equivalent to SRFI-43’s vector-for-each. See Section 11.12 [Vector library (Legacy)], page 717.

6.13.2 Uniform vectors
Uniform vectors, or homogeneous numeric vectors, are a special type of vectors whose elements are of the same numeric type. It was introduced originally as SRFI-4, revised by SRFI-160, and now a part of R7RS large (as scheme.vector.@).

The @ part is actually one of the following tags, indicating the type of elements:

- **u8**  Unsigned 8-bit integer - an exact integer between 0 and 255.
- **s8**  Signed 8-bit integer - an exact integer between -128 and 127.
- **u16** Unsigned 16-bit integer - an exact integer between 0 and 65535.
- **s16** Signed 16-bit integer - an exact integer between -32768 and 32767.
- **u32** Unsigned 32-bit integer - an exact integer between 0 and \( 2^{32} - 1 \).
- **s32** Signed 32-bit integer - an exact integer between \( -(2^{31}) \) and \( 2^{31} - 1 \).
- **u64** Unsigned 64-bit integer - an exact integer between 0 and \( 2^{64} - 1 \).
- **s64** Signed 64-bit integer - an exact integer between \( -(2^{63}) \) and \( 2^{63} - 1 \).
- **f16** 16-bit floating point number (10-bit mantissa and 5-bit exponent), as inexact real.
- **f32** IEEE single-precision floating point number as inexact real.
- **f64** IEEE double-precision floating point number as inexact real.
- **c32** Inexact complex, consists of a pair of 16-bit floating point numbers.
- **c64** Inexact complex, consists of a pair of IEEE single-precision floating point numbers.
- **c128** Inexact complex, consists of a pair of IEEE double-precision floating point numbers.

There are some advantages of using uniform vectors over normal (heterogeneous) vectors. It may be more compact than the normal vectors. Some operations (especially Gauche’s extension of vector arithmetic operations) can bypass type check and conversion of individual elements, thus be more efficient. And it is much easier and efficient to communicate with external libraries that require homogeneous array of numbers; for example, OpenGL binding of Gauche uses uniform vectors extensively.

Gauche has only a handful primitive operations on uniform vectors as a built-in, but the gauche.uvector module, or scheme.vector.@ module ((scheme vector @) library in R7RS programs), provide a comprehensive set of operations. See Section 9.38 [Uniform vector library], page 556, and see Section 10.3.3 [R7RS uniform vectors], page 602.
Uvector classes

<uvector> [Abstract Class]

The base class of uniform vector classes. It inherits <sequence> (see Section 9.31 [Sequence framework], page 510).

@vector [Builtin Class]

{gauche.uvector} A class for @vector, where @ is one of the uvector tags (u8, s8, . . .). It inherits <uvector>.

It implements sequence protocol (see Section 9.31 [Sequence framework], page 510), so you can convert a sequence of real numbers into a uvector using coerce-to, if every elements is valid for the vector.

(use gauche.sequence)
(coerce-to <u8vector> '(1 2 3)) ⇒ #u8(1 2 3)

Uvector literals

#u8(n ...)
#s8(n ...)
#u16(n ...)
#s16(n ...)
#u32(n ...)
#s32(n ...)
#u64(n ...)
#s64(n ...)
#f16(n ...)
#f32(n ...)
#f64(n ...)
#c32(n ...)
#c64(n ...)
#c128(n ...)

[Reader Syntax]

Denotes a literal homogeneous vector.
(Note: R7RS bytevector is the same as u8vector, and can be written as #u8(...).)

#s8(3 -2 4)
#u32(4154 88357 2 323)
#f32(3.14 0.554525 -3.342)

Gauche also supports SRFI-207 (String-notated bytevector) style bytevector literal. It is a prefix #u8 followed by a string literal, in which only ASCII characters and escaped single-octets (i.e. \xNN; where N is a hexadecimal digit) are allowed. It allows to naturally use bytevectors as a bytestring. See Section 11.47 [String-notated bytevectors], page 776, for operations on bytestrings.

#u8"ABC\x00;D" ⇒ #u8(65 66 67 0 68)

Uvector constructors

make-s8vector len :optional fill
make-s8vector len :optional fill
make-u8vector len :optional fill
make-s16vector len :optional fill
make-u16vector len :optional fill
make-s32vector len :optional fill
make-u32vector len :optional fill

[Function]
Chapter 6: Core library

```scheme
make-s64vector len :optional fill
make-u64vector len :optional fill
make-f16vector len :optional fill
make-f32vector len :optional fill
make-f64vector len :optional fill
make-c32vector len :optional fill
make-c64vector len :optional fill
make-c128vector len :optional fill

[R7RS vector.@] Constructs a @vector of length `len`. The elements are initialized by a number `fill`. For exact integer vectors, `fill` must be an exact integer and in the valid range. If `fill` is omitted, the content of the vector is undefined.

(make-u8vector 4 0) ⇒ #u8(0 0 0 0)
```

**Uvector generic operations**

`uvector? obj`  [Function]
Returns `#t` iff `obj` is one of the uniform vectors. See below for predicates for specific type of uvector.

`uvector-length uv`  [Function]
Returns the length (the number of elements) of uvector `uv`. An error is raised if `uv` is not a uvector.

Type specific length procedures are provided in `scheme.vector.@` and `gauche.uvector` (see Section 9.38 [Uniform vector library], page 556).

To get the size of the binary data the content of the uvector actually occupies, use `uvector-size` in `gauche.uvector`.

`uvector-ref uv k :optional fallback`  [Function]
Generic uvector accessor. Returns `k`-th element of a uniform vector `uv`. If `k` is out-of-range, `fallback` is returned if provided, or an error is thrown otherwise.

This is handy to write a generic code that works on any kind of uniform vector, but this is slower than the specific versions. Gauche’s compiler recognizes the specific versions of referencer and generate very efficient code for them, while this generic version becomes a normal procedure call. In inner-loop it can make a big difference.

See below for the type-specific accessors.

`(setter uvector-ref)` is `uvector-set!`.

`uvector-set! uv k val :optional clamp`  [Function]
Generic uvector setter. Mutate `k`-th element of uvector `uv` with `val`. An error is thrown if `k` is out-of-range, or `uv` is immutable.

Optional `clamp` argument specifies the behavior when `val` is out of valid range. It can be `#f` or one of the symbols `low`, `high`, or `both`. See Section 9.38 [Uniform vector library], page 556, for the meanings of the clamp argument. The default is `#f`, which raises an error on out-of-range value.

**Uvector type-specific operations**

Type-specific predicates, accessors and modifiers are provided in the core library; all the rest are in `scheme.vector.@` or `gauche.uvector` (see Section 9.38 [Uniform vector library], page 556).

`@vector? obj`  [Function]
[R7RS vector.@] Returns `#t` iff `obj` is a @vector, `#f` otherwise. The `@` part is one of the uvector tags (u8 etc.).
@vector-ref vec k :optional fallback

[R7RS vector.@] Returns the k-th element of @vector vec. The @ part is one of the uvector tags (u8 etc.).

If the index k is out of the valid range, an error is signaled unless an optional argument fallback is given; in that case, fallback is returned.

Note that the generic function ref can be used as well, if you import gauche.collection.

```
(u16vector-ref '#u16(111 222 333) 1) ⇒ 222
```

```
(use gauche.collection)
(ref '#u16(111 222 333) 1) ⇒ 222
```

Setter of @vector-ref is @vector-set!.

```
(define v (u8vector 1 2 3))
(set! (u8vector-ref v 1) 99)
```

```
v ⇒ #u8(1 99 3)
```

@vector-set! vec k n :optional clamp

[R7RS vector.@] Sets a number n to the k-th element of @vector vec. The @ part is one of the uvector tags (u8 etc.).

Optional clamp argument specifies the behavior when n is out of valid range. It can be #f or one of the symbols low, high, or both. See Section 9.38 [Uniform vector library], page 556, for the meanings of the clamp argument. The default is #f, which raises an error on out-of-range value.

Note that the setter of the generic function ref can be used as well, if you import gauche.collection.

```
(let ((v (s32vector -439 852 8933)))
  (s32vector-set! v 1 4)
  v)
⇒ #s32vector(-439 4 8933)
```

```
(use gauche.collection)
(let ((v (s32vector -439 852 8933)))
  (set! (ref v 1) 4)
  v)
⇒ #s32vector(-439 4 8933)
```

Conversion between strings and uvectors

Two sets of conversion procedures are provided. The first set is to convert strings to u8/s8 vector and vice versa, taking multibyte representation of a string as a byte vector. The second set is to convert strings to u32/s32 vector and vice versa, regarding a string as a sequence of Unicode codepoints.

For portable Scheme code, SRFI-181 provides conversion between strings and bytevectors, specifying character encodings (see Section 11.40.2 [Transcoded ports], page 761).

string->s8vector string :optional start end immutable?

string->u8vector string :optional start end immutable?

Returns an s8vector or u8vector whose byte sequence is the same as the internal representation of the given string. Optional range arguments start and end specifies the character position (not the byte position) inside string to be converted.
By default, the content of the string is copied to a newly created mutable uvector. However, if a true value is given to the optional *immutable?* argument, the result is an immutable uvector, and it may avoid copying the string body (note that in Gauche, the body of string is immutable; `string-set!` creates a new body, so changing the original string won’t affect the uvector created by `string->u8vector` with *immutable?* flag.)

These procedures are useful when you want to access byte sequence of the string randomly.

```
(string->u8vector "abc")  ⇒  #u8(97 98 99)
```

```
(string->u8vector "very large string .... " 0 -1 #t)
⇒  #u8(...) ; immutable, sharing content with the original string
```

```
(string->s8vector! target tstart string :optional start end)
```

```
(string->u8vector! target tstart string :optional start end)
```

*Target* must be an s8vector or a u8vector, respectively. *Target* must be mutable. Like copies the raw byte representation of *string* into *target* beginning from index *tstart*.

Returns *target*.

```
(let ((target (make-u8vector 10 0)))
  (string->u8vector! target 3 "abcde"))
⇒  #u8(0 0 0 97 98 99 100 101 0 0)
```

```
(s8vector->string vec :optional start end terminator)
```

```
(u8vector->string vec :optional start end terminator)
```

Converts a byte sequence in s8vector or u8vector to a string that has the same byte sequence. Optional range arguments *start* and *end* specifies the byte position in *vec* to be converted.

The optional *terminator* argument can be an exact integer or #f (default). If it is an exact integer, and it appears in *vec*, the string terminates right before it. For example, you can give 0 as *terminator* to read a NUL-terminated string from a buffer.

```
(u8vector->string '#u8(65 66 0 67 68) 0 5)  ⇒  "AB\0CD"
(u8vector->string '#u8(65 66 0 67 68) 0 5 0)  ⇒  "AB"
```

Note that these procedure may result an incomplete string if *vec* contains a byte sequence invalid as the internal encoding of the string.

```
(string->s32vector string :optional start end endian)
```

```
(string->u32vector string :optional start end endian)
```

Returns an s32vector or u32vector whose elements are the internal codes of the characters in the string. Optional range arguments *start* and *end* specifies the *character position* inside *string* to be converted.

The optional *endian* argument specifies the endianness to store codepoint value into the uvector. When omitted or #f, we use the platform’s native endianness. You can give a symbol big-endian, big, little-endian, little or arm-little-endian to use a specific endianness. Note that when you access the resulting uvector, the platform’s native endianness is used.

These procedures are useful when you want to access the characters in the string randomly.
The optional $endian$ argument specifies the endianness to store codepoint value into the uvector. When omitted or $\#f$, we use the platform’s native endianness. You can give a symbol $big-endian$, $big$, $little-endian$, $little$ or $arm-little-endian$ to use a specific endianness. Note that when you access the resulting uvector, the platform’s native endianness is used.

\[
\begin{align*}
\text{s32vector->string} & \text{ vec :optional start end terminator endian} \\
\text{u32vector->string} & \text{ vec :optional start end terminator endian}
\end{align*}
\]

Without $start$ and $end$, these procedures work like this:

\[(\lambda (\text{vec}) (\text{map-to } <\text{string}> \text{ integer->char vec}))\]

Optional range arguments $start$ and $end$ limits the range of conversion between them.

The optional $terminator$ argument must be an exact integer or $\#f$ (default). If an integer is given, and the integer is found in the input, the output string terminates right before it.

The optional $endian$ argument specifies the endianness to store codepoint value into the uvector. When omitted or $\#f$, we use the platform’s native endianness. You can give a symbol $big-endian$, $big$, $little-endian$, $little$ or $arm-little-endian$ to use a specific endianness.

\[(\text{u32vector->string } \#u32(65 66 0 67 68) 0 5 0) \Rightarrow "AB"\]

### 6.13.3 Bitvectors

A bitvector is a sequence of bits. Each bit can be considered either an exact integer 0/1, or a boolean values $\#t/\#f$. In the former view, it is similar to a uniform vector, but it has the interface sufficiently different from uvectors and we provided it as a disjoint type.

Gauche provides a handful of procedures in the core, and $\text{gauche.bitvector}$ provides comprehensive utilities. See Section 9.3 [Bitvector utilities], page 374, for the details. For the portable programs, SRFI-178 defines bitvector library, which is a subset of what Gauche provides (see Section 11.38 [Bitvector library], page 756).

\textbf{<bitvector>} [Builtin class]

Bitvector class. Inherits $\text{<sequence>}$, so generic sequence operaitions can be used. (Generic $\text{ref}$ uses $\text{bitvector-ref/int}$, for it matches the external representation of a bitvector.)

\textbf{#*b...} [Reader Syntax]

[SRFI-178] A bitvector literal is $\#*$ followed by zero or more binary digits 0 or 1.

\begin{align*}
\#*10010010 & \quad \text{bitvector of length 8} \\
\#* & \quad \text{bitvector of length 0}
\end{align*}

A bitvector literal is delimited by one of delimiter character or an EOF.

\begin{align*}
\#*10010abc & \quad \text{error} \\
\#*10001(a b c) & \quad \text{a bitvector, followed by a list}
\end{align*}

Note: With this rule, $\#*\ldots\"$ should be read as a zero-length bitvector followed by a string, for $\"$ is a delimiter. However, Gauche used that syntax for incomplete strings (our overlook!). Since incomplete string literals is rare (incomplete strings are something that unexpectedly happen in the practical situation, but not to be used actively), we changed incomplete string literals to $\#*\ldots\"$ since 0.9.10 (see Section 6.11.10 [Incomplete strings], page 182).

For the backward compatibility, the current version reads $\#*\ldots\"$ as an incomplete string. If the reader lexical mode is $\text{warn-legacy}$ (see Section 6.21.7.2 [Reader lexical mode], page 272), such literals are warned. We’ll gradually migrate to make $\#*\ldots\"$ read as a bytevector followed by a string.
Chapter 6: Core library

```
bit->integer bit
bit->boolean bit
[SRFI-178] Many bitvector operations can accept bit as a boolean (#f/#t) or an exact integer (0/1). These are utility procedures to obtain desired type. The bit argument must be either one of #f, #t, 0 or 1. They return 0/1 and #f/#t, respectively. An error is signalled if bit is other than those values.

bitvector b . . .
[SRFI-178] Creates and returns a bitvector whose elements are b . . . . Each argument must be a bit (boolean or 0 or 1).

(make-bitvector 5 #f) ⇒ #**00000
(make-bitvector 7 1) ⇒ #**1111111

bitvector? obj
[SRFI-178] Return #t iff obj is an bitvector.

list->bitvector lis
[SRFI-178] Lis must be a list of bits (0, 1 or booleans). Returns a bitvector whose elements consist of elements of lis.

(string->bitvector "#*1010001") ⇒ #*1010001
(string->bitvector "#*1001020") ⇒ #f

Note that this isn’t a sequence-conversion, but rather a conversion from external representation.

bitvector->string bv
[SRFI-178] Convert a bitvector bv to a string representation #*b....

Note that this isn’t a sequence-conversion, but rather a conversion to external representation.

bitvector-ref/int bv k :optional fallback
bitvector-ref/bool bv k :optional fallback
[SRFI-178+] Retrieves the k-th bit of a bitvector bv as an integer or a boolean value, respectively. If k is out of range, fallback is returned if it is given, or an error is raised otherwise. The fallback argument is Gauche’s extension.

(\texttt{(bitvector-ref/int \#*1010001 0)}) ⇒ 1
(\texttt{(bitvector-ref/bool \#*1010001 0)}) ⇒ #t

If you use a universal accessor \texttt{ref/\textasciitilde{}}, it returns the bit value as an integer (see Section 6.15.2 [Universal accessor], page 219).

(\texttt{~ \#*11001001 1)}) ⇒ 1
```
Function \texttt{bitvector-set! \textit{bv} \textit{k} \textit{bit}} \hfill [SRFI-178] Sets the \textit{k}-th bit of a bitvector \textit{bv} with \textit{bit}, which must be either one of 0, 1, \#f or \#t. An error is raised if \textit{k} is out of range.

This procedure is set as the setter of \texttt{bitvector-ref/int} and \texttt{bitvector-ref/bool}. Since a bitvector is a sequence, you can also use \texttt{(setter ref)/(setter ~)}:

\begin{verbatim}
(rlet1 z (make-bitvector 5 0)
  (set! (~ z 2) #t))
⇒ #00100
\end{verbatim}

Function \texttt{bitvector-copy \textit{bv} :optional \textit{start} \textit{end}} \hfill [SRFI-178] Returns a copy of a bitvector \textit{bv}. If optional \textit{start} and \textit{end} indexes are given, the copy is limited in that range, where \textit{start} is inclusive and \textit{end} is exclusive.

\begin{verbatim}
(bitvector-copy #*101001000 3) ⇒ #*001000
(bitvector-copy #*100101000 2 7) ⇒ #*01010
\end{verbatim}

Function \texttt{bitvector-copy! \textit{target} \textit{tstart} \textit{src} :optional \textit{sstart} \textit{send}} \hfill [SRFI-178] Copy a bitvector \textit{src} into a mutable bitvector \textit{target} starting from \textit{tstart}, mutating \textit{target}. Optional \textit{sstart} and \textit{send} delimits the range in \textit{src}.

\begin{verbatim}
(rlet1 v (make-bitvector 10 0)
  (bitvector-copy! v 3 #*101101110 2 6))
⇒ #*0001101000
\end{verbatim}

Function \texttt{bitvector-any-value? \textit{bv} \textit{bit} :optional \textit{start} \textit{end}} \hfill [Function] Check if any or every bit of a bitvector \textit{bv} is \textit{bit}, respectively. You can pass a boolean value or an exact integer 0 or 1 as \textit{bit}.

Optional \textit{start/end} arguments limit the range of bitvector to look at.

\begin{verbatim}
;; See if at least one '1'
(bitvector-any-value? #*0001000 1) ⇒ #t
(bitvector-any-value? #*000000 1) ⇒ #f

;; See if at least one '0'
(bitvector-any-value? #*110111 #f) ⇒ #t
(bitvector-any-value? #*11111 0) ⇒ #f

;; See if all bits are '1'
(bitvector-every-value? #*11111 #t) ⇒ #t
(bitvector-every-value? #*00000 1) ⇒ #f

;; See if all bits are '0'
(bitvector-every-value? #*11111 #f) ⇒ #f
(bitvector-every-value? #*00000 0) ⇒ #t
\end{verbatim}

6.13.4 Weak vectors

A weak pointer is a reference to an object that doesn’t prevent the object from being garbage-collected. Gauche provides weak pointers as a \textit{weak vector} object. A weak vector is like a vector of objects, except each object can be garbage collected if it is not referenced from objects other than weak vectors. If the object is collected, the entry of the weak vector is replaced for \#f.

\begin{verbatim}
gosh> (define v (make-weak-vector 1))
v
gosh> (weak-vector-ref v 0)
\end{verbatim}
gosh> (weak-vector-set! v 0 (cons 1 1))
#<undef>
gosh> (weak-vector-ref v 0)
(1 . 1)
gosh> (gc)
#<undef>
gosh> (gc)
#<undef>
gosh> (weak-vector-ref v 0)
#f

See Section 10.3.17 [R7RS ephemeron], page 639, for R7RS-large way of weak pointers.

[Function] make-weak-vector size
Creates and returns a weak vector of size size.

[Function] weak-vector-length wvec
Returns the length of a weak vector wvec.

[Function] weak-vector-ref wvec k :optional fallback
Returns k-th element of a weak vector wvec.

By default, weak-vector-ref signals an error if k is negative, or greater than or equal to the size of wvec. However, if an optional argument fallback is given, it is returned for such case.

If the element has been garbage collected, this procedure returns fallback if it is provided, #f otherwise.

With gauche.sequence module, you can also use a method ref.

[Function] weak-vector-set! wvec k obj
Sets k-th element of the weak vector wvec to obj. It is an error if k is negative or greater than or equal to the size of wec.

6.14 Dictionaries

A dictionary is a data structure that associates key to value. Gauche provides hashtables (see Section 6.14.1 [Hashtables], page 206) and treemaps (see Section 6.14.2 [Treemaps], page 212) as the built-in dictionaries. Some additional libraries provide more dictionary-type data structures.

A generic interface is defined as a dictionary framework (see Section 9.10 [Dictionary framework], page 428), by which you can use dictionaries without knowing its details.

R7RS also defines an abstract dictionary interface as mapping; see Section 10.3.20 [R7RS mappings], page 652, for the details.

6.14.1 Hashtables

R7RS-large defines hashtable (scheme.hash-table module, see Section 10.3.7 [R7RS hash tables], page 618) but its API is not completely consistent with Gauche’s original hashtables and other native APIs.
Rather than mixing different flavor of APIs, we keep Gauche’s native API consistent, and provide R7RS procedures that are inconsistent with aliases—specifically, those procedures are suffixed with -r7 in gauche module. For portable programs, you can import scheme.hash-table to get R7RS names.

[hash-table]
[Builtin Class]
Hash table class. Inherits <collection> and <dictionary>.
Gauche doesn’t provide immutable hash tables for now. (If you need immutable maps, see Section 12.17 [Immutable map], page 834).

Hash table properties

hash-table? obj [Function]
[R7RS hash-table] Returns #t iff obj is a hash table.

hash-table-mutable? ht [Function]
[R7RS hash-table] Returns #t iff a hash table ht is mutable. Gauche doesn’t have immutable hash tables, so this procedure always returns #t for any hash tables.

hash-table-comparator ht [Function]
Returns a comparator used in the hashtable ht.

hash-table-type ht [Function]
This is an old API, superseded by hash-table-comparator.
Returns one of symbols eq?, eqv?, equal?, string=?, general, indicating the type of the hash table ht.

hash-table-num-entries ht [Function]
hash-table-size ht [Function]
[R7RS hash-table] Return the number of entries in the hash table ht. R7RS name is hash-table-size.

Hash table constructors and converters

make-hash-table :optional comparator [Function]
[R7RS+ hash-table] Creates a hash table. The comparator argument specifies key equality and hash function using a comparator (see Section 6.2.4 [Basic comparators], page 116). If omitted, eq-comparator is used. Note that in R7RS, comparator argument can’t be omitted. As Gauche’s extension, the comparator argument can also be one of the symbols eq?, eqv?, equal? or string=??. If it is one of those symbols, eq-comparator, eqv-comparator, equal-comparator and string-comparator will be used, respectively.
The comparator must have hash function, of course. See Section 6.2.3 [Hashing], page 113, for the built-in hash functions. In general, comparators derived from other comparators having hash functions also have appropriate hash functions.

hash-table-from-pairs comparator key&value . . . [Function]
Constructs and returns a hash table from given list of arguments. The comparator argument is the same as of make-hash-table. Each key&value must be a pair, and its car is used as a key and its cdr is used as a value.
Note: This is called hash-table by 0.9.5. R7RS introduced a procedure with the same name, but different interface. We see R7RS version makes more sense, so we’ll eventually switch to it, but the transition will take long time. The R7RS interface is available as
hash-table-r7, and we urge you to use it in the new code, and replace existing hash-table with hash-table-from-pairs.

\[
\text{(hash-table-from-pairs 'eq? '(a . 1) '(b . 2))}
\equiv
\text{(rlet1 h (make-hash-table 'eq?)}
\text{(hash-table-put! h 'a 1)}
\text{(hash-table-put! h 'b 2))}
\]

hash-table comparator key&value . . .  [Function]
An alias of hash-table-from-pairs above. R7RS introduced the same name procedure with different interface (see hash-table-r7 below), and we’d like to switch to it in future. For now, use either hash-table-from-pairs or hash-table-r7, or import scheme.hash-table and write in R7RS.

hash-table-r7 comparator args . . .  [Function]
Create and returns a hash table using comparator. The args . . . are the contents, alternating keys and values.

This is defined as hash-table in R7RS scheme.hash-table (see Section 10.3.7 [R7RS hash tables], page 618).

\[
\text{(hash-table-r7 'eq? 'a 1 'b 2)}
\equiv
\text{(rlet1 h (make-hash-table 'eq?)}
\text{(hash-table-put! h 'a 1)}
\text{(hash-table-put! h 'b 2))}
\]

Note: An R7RS compliant implementation of hash-table may return an immutable hash table. Since Gauche doesn’t have immutable hash tables (we have immutable maps instead; see Section 12.17 [Immutable map], page 834), we return mutable hash tables. However, the portable program should refrain from mutating the returned hash tables.

hash-table-unfold p f g seed comparator :rest args  [Function]
[R7RS hash-table] Constructs and returns a new hash table with those repetitive steps. Each iteration keeps the current seed value, whose initial value is seed.
1. Apply a stop predicate p to the current seed value. If it returns a true value, stop.
2. Apply a value producer f to the current seed value. It must return two values, which are used as a key and the corresponding value, of the hash table.
3. Apply a next procedure g to the current seed value. The value it returns becomes the next seed value.

hash-table-copy ht :optional mutable?  [Function]
[R7RS hash-table] Returns a new copy of a hash table ht.
R7RS defines this procedure to return an immutable hash table if the implementation supports one, unless the optional mutable? argument is provided and not false. Gauche doesn’t have immutable hash tables so it ignores the optional argument and always returns a mutable hash table. But when you write a portable programs, keep it in mind.

hash-table-empty-copy ht  [Function]
[R7RS hash-table] Returns a new mutable empty hash table that has the same properties as the given hash table ht.

alist->hash-table alist :optional comparator  [Function]
[R7RS+ hash-table] Creates and returns a hash table that has entries of each element in alist, using its car as the key and its cdr as the value. The comparator argument is the same as in make-hash-table. The default value of comparator is eq-comparator.
R7RS doesn’t allow to omit comparator.

hash-table->alist hash-table

[Function]
(hash-table-map h cons)

Hash table lookup and mutation

hash-table-get ht key :optional default

Search key from a hash table ht, and returns its value if found. If the key is not found in the table and default is given, it is returned. Otherwise an error is signaled.

hash-table-put! ht key value

Puts a key key with a value value to the hash table ht.

ref (ht <hash-table>) key :optional default

(setter ref) (ht <hash-table>) key value

Method versions of hash-table-get and hash-table-put!.

hash-table-ref ht key :optional failure success

[Function]

[R7RS hash-table] This is R7RS way to look up a hash table.

Look up a value associated to the key in the table ht, then pass it to a procedure success, and returns its value. If success is omitted, an identity function is used.

If there’s no association for key in ht, a thunk failure is called and its result is returned. The default value of failure throws an error.

It is more general than Gauche’s hash-table-get, but if you need to simply return a fallback value in case of failure, you need to wrap it with a clojure, which is annoying. In R7RS, you can use hash-table-ref/default below.

hash-table-ref/default ht key default

[Function]

[R7RS hash-table] Looks up key in a hash table ht and returns the associated value. If there’s no key in the table, returns default.

This is same as Gauche’s hash-table-get, except that default is not optional. We provide both, for hash-table-ref is short and handy.

hash-table-set! ht args . . .

[Function]

[R7RS hash-table] This is R7RS version to put associations into a hash table. The args . . . is a list of alternating keys and values; so, unlike Gauche’s hash-table-put!, you can insert more than one associations at once. It is an error if args . . . have odd number of arguments.

(hash-table-set! ht ‘a 1 ’b 2)

≡

(begin (hash-table-put! ht ’a 1)

(hash-table-put! ht ’b 2))

hash-table-intern!-r7 ht key failure

[Function]

This is defined in R7RS as hash-table-intern!. We add -r7 suffix to remind that it takes a failure thunk, which is consistent with R7RS hash-table interface but not Gauche’s way.

Lookup key in ht. If there’s already an entry, it just returns the value. Otherwise, it calls a thunk failure, and insert the association of key and the return value of failure into ht, and returns the value.

hash-table-exists? ht key

hash-table-contains? ht key

[Function]

[R7RS hash-table] Returns #t if a hash table ht has a key key.

R7RS name is hash-table-contains?.

hash-table-delete! ht key
[Function]
Deletes an entry that has a key key from the hash table ht. Returns #t if the entry has exist, or #f if the entry hasn’t exist. The same function is called hash-table-remove! in STk (except that it returns an undefined value); I use ‘delete’ for consistency to SRFI-1, SRFI-13 and other parts of the libraries.

Note: This is different from R7RS hash-table-delete!, so we provide R7RS interface with an alias hash-table-delete!-r7.

hash-table-delete!-r7 ht key . . .
[Function]
Delets entries that have key . . . from the hash table ht. The key which isn’t in ht has no effect. Returns the number of entries actually deleted.

This is called hash-table-delete! in R7RS, and so as in scheme.hash-table. We provide this under different name, for Gauche’s hash-table-delete! returns a boolean value.

hash-table-clear! ht
[R7RS hash-table] Removes all entries in the hash table ht.

hash-table-push! ht key value
[Function]
Conses value to the existing value for the key key in the hash table ht and makes it the new value for key. If there’s no entry for key, an entry is created with the value (list value).

Works the same as the following code, except that this function only looks up the key once, thus it’s more efficient.

(let ((tmp (proc (hash-table-get ht key default))))
  (hash-table-put! ht key
tmp)

hash-table-pop! ht key :optional default
[Function]
Looks for the value for the key key in the hash table ht. If found and it is a pair, replaces the value for its cdr and returns car of the original value. If no entry for key is in the table, or the value is not a pair, the table is not modified and the procedure returns default if given, or signals an error otherwise.

During the operation the key is looked for only once, thus runs efficiently.

Note: R7RS has hash-table-pop! but its totally different. We provide R7RS version as an alias hash-table-pop!-r7

hash-table-pop!-r7 ht
[Function]
Removes one arbitrary entry from ht, and returns the removed entry’s key and value as two values. If ht is empty, an error is thrown.

This is called hash-table-pop! in R7RS, and so as in scheme.hash-table.

hash-table-update! ht key proc :optional default
[Function]
A more general version of hash-table-push! etc. It works basically as the following code piece, except that the lookup of key is only done once.

(let ((tmp (proc (hash-table-get ht key default))))
  (hash-table-put! ht key tmp)
tmp)

For example, when you use a hash table to count the occurrences of items, the following line is suffice to increment the counter of the item, regardless of whether item has already appeared or not.

(hash-table-update! ht item (cut + 1 <> 0))

R7RS provides hash-table-update! with different interface, so we provide R7RS version as an alias hash-table-update!-r7.
hash-table-update!-r7  ht  key  updater  :optional  failure  success

This is R7RS version of hash-table-update!. With no optional arguments, it works like Gauche’s hash-table-update!. But in practice you often needs to specify the behavior when key hasn’t been in ht, in which case R7RS differs from Gauche.

The R7RS version works like this but potentially more efficiently:

(hash-table-put! ht key (updater (hash-table-ref ht key failure success)))

hash-table-update!/-default  ht  key  updater  default

[R7RS hash-table] This is the same as Gauche’s hash-table-default!, except that the default value can’t be omitted.

Hash table scanners

hash-table-for-each  ht  proc
hash-table-map  ht  proc

A procedure proc is called with two arguments, a key and its associated value, over all the entries in the hash table ht.

hash-table-fold  ht  kons  knil

For all entries in the hash table ht, a procedure kons is called with three arguments; a key, its associated value, and the previous return value of kons. The first call of kons receives knil as the third argument. The return value of the last call of kons is returned from hash-table-fold.

hash-table-find  ht  pred  :optional  failure

Apply pred with each key and value in the hash table ht. Once pred returns a true value, that return value is immediately returned from hash-table-find. If no key-value satisfies pred, a thunk failure is invoked and its result is returned. If failure is omitted, (lambda () #f) is assumed.

Note: The convention starting from SRFI-1 is that *-find returns an item in the collection that satisfy the predicate, while *-any returns a non-false value the predicate returns. SRFI-125 broke the convention. The justification given in SRFI-125 discussion was that the “any” semantics is strictly upper-compatible to the “find” semantics so we can combine two. So far, though, SRFI-125 is the only exception of this convention.

;;; Find if hash tables ha and hb has a common key.
(hash-table-find ha (^[k v] (hash-table-exists? hb k)))

hash-table-keys  ht
hash-table-values  ht

Returns all the keys or values of hash table ht in a list, respectively.

Hash table as sets

hash-table-compare-as-sets  ht1  ht2  :optional  value-=?  fallback

A hash table can be viewed as a set of pairs of key and value. This procedure compares two hash tables ht1 and ht2 as such sets.

The key comparators of two tables must match (in terms of equal? of the comparators). Otherwise, an error is signaled.

Two elements of the set are equal to each other iff their keys match with the equality predicate of the key comparator, and their values match with value=? procedure. If omitted, equal? is used for value=?

There can be four cases.

• If ht1 is a pure subset of ht2, returns -1 (ht1 is smaller than ht2).
• If \( \text{ht2} \) is a pure subset of \( \text{ht1} \), returns 1 (\( \text{ht1} \) is greater than \( \text{ht2} \)).
• If \( \text{ht1} \) and \( \text{ht2} \) contains exactly the same elements, returns 0 (\( \text{ht1} \) equals to \( \text{ht2} \)).
• Neither \( \text{ht1} \) nor \( \text{ht2} \) is a subset of another. In this case, \text{fallback} is returned if it is given, or an error is thrown.

\textit{hash-table=} \textit{value-cmpr} \textit{ht1} \textit{ht2}  
[Function]

[R7RS hash-table] This also compares two hash tables \( \text{ht1} \) and \( \text{ht2} \) as sets, and returns true iff two are the same. That is, every element in \( \text{ht1} \) is also in \( \text{ht2} \) and vice versa.

Two element are the same iff their keys are the same in terms of the equality predicate of the tables’ key comparator, and their values are the same in terms of the equality predicate of a comparator \( \text{value-cmpr} \).

It is an error if \( \text{ht1} \) and \( \text{ht2} \) has different key comparators. See also \textit{hash-table-compare-as-sets} above.

\textit{hash-table-union!} \textit{ht1} \textit{ht2}  
[Function]

\textit{hash-table-intersection!} \textit{ht1} \textit{ht2}  
[Function]

\textit{hash-table-difference!} \textit{ht1} \textit{ht2}  
[Function]

\textit{hash-table-xor!} \textit{ht1} \textit{ht2}  
[Function]

[R7RS hash-table] Perform set operations on two hashtables \( \text{ht1} \) and \( \text{ht2} \), and modify \( \text{ht1} \) to store the result. Note that these procedures only look at the keys for operation; if the values of the same key differ between \( \text{ht1} \) and \( \text{ht2} \), the value in \( \text{ht1} \) is taken.

• The union operation picks each entry that is in at least one of \( \text{ht1} \) or \( \text{ht2} \).
• The intersection operation picks each entry that is both in \( \text{ht1} \) and \( \text{ht2} \).
• The difference operation picks each entry that is in \( \text{ht1} \) but not in \( \text{ht2} \).
• The xor operation picks each entry that is in only one of \( \text{ht1} \) or \( \text{ht2} \), but not in both.

6.14.2 Treemaps

\textit{<tree-map>}  
[Builtin Class]

Tree map class. Tree maps are a data structure that maps key objects to value objects. It’s like hash tables except tree maps uses balanced tree internally. Insertion and lookup is O(log n).

Unlike hashtables, a tree map keeps the order of the keys, so it is easy to traverse entries in the order of keys, to find minimum/maximum keys, or to find a key closest to the given value.

The \textit{<tree-map>} class inherits \textit{<sequence>} and \textit{<ordered-dictionary>}.

\textit{make-tree-map} :optional \textit{comparator}  
[Function]

\textit{make-tree-map} \textit{key=} ? \textit{key<}  
[Function]

Creates and returns an instance of \textit{<tree-map>}. The keys are compared by \textit{comparator}, whose default is \textit{default-comparator}. The comparator must have a comparison procedure, for we need a total order in the keys. See Section 6.2.4 [Basic comparators], page 116, for the details.

For the backward compatibility, \textit{make-tree-map} also accepts a procedure as a \textit{comparator}; the procedure must take two keys and returns either -1, 0, or 1, depending on whether the first key is less than, equal to, or greater than the second key, respectively. In other words, it is a comparison procedure of a comparator.

The second form of \textit{make-tree-map} is also for the backward compatibility; it takes two procedures, each must be a procedure that takes two keys; the first one returns \texttt{#t} iff two keys are equal, and the second one returns \texttt{#t} iff the first key is strictly smaller than the second.
tree-map-comparator tree-map
  Returns the comparator used in the tree map.

tree-map-copy tree-map
  Copies and returns tree-map. Modification on the returned tree doesn’t affect the original tree.

tree-map-empty? tree-map
  Returns #t if tree-map doesn’t have any elements, or #f otherwise.

tree-map-num-entries tree-map
  Returns the number of elements in tree-map.

tree-map-exists? tree-map key
  Returns #t if tree-map has an entry with key, or #f otherwise.

tree-map-get tree-map key :optional fallback
  Looks for key in tree-map. If the entry is found, returns a value corresponding to the key. Otherwise, returns fallback if it is provided, or signals an error.

tree-map-put! tree-map key value
  Inserts an entry with a key and corresponding value into tree-map. If there already exists an entry with a key which is equivalent (under key=?), the entry is modified to have value.

tree-map-delete! tree-map key
  Deletes an entry with key from tree-map if such an entry exists, and returns #t. If tree-map doesn’t have such an entry, #f is returned.

tree-map-clear! tree-map
  Removes all entries in tree-map.

tree-map-update! tree-map key proc :optional fallback
  A generalized version of tree-map-push! etc. It works like the following code, except that searching for the key is done only once.

  (let ((tmp (proc (tree-map-get tree-map key fallback))))
  (tree-map-put! tree-map key tmp)
  tmp)

tree-map-push! tree-map key value
  Looks for an entry with key in tree-map. If it exists, the procedure conses value to the original value and makes it as a new value. Otherwise, the procedure creates a new entry for the key and makes (list value) its value.

tree-map-pop! tree-map key :optional fallback
  Looks for an entry with key in tree-map. If it exists and its value is a pair, then the procedure updates its value with cdr of the original value, and returns car of the original entry. If such an entry does not exist, or has a non-pair value, the procedure doesn’t modify tree-map and returns fallback if it is given, otherwise reports an error.

tree-map-min tree-map
  Returns a pair of a key and its value with the minimum or maximum key, respectively. If tree-map is empty, #f is returned.
tree-map-pop-min! tree-map  [Function]
Looking for an entry with minimum or maximum key, respectively, then deletes the entry from

  \( \text{tree-map} \) and returns a pair of the key and its value of the original entry. If

  \( \text{tree-map} \) is empty, \#f is returned.

\[ \text{tree-map-fold} \quad \text{tree-map proc seed} \]  [Function]
\[ \text{tree-map-fold-right} \quad \text{tree-map proc seed} \]  [Function]
Iterate over elements in \( \text{tree-map} \), applying \( \text{proc} \) which has a type \((\text{key}, \text{value}, \text{seed}) \) ->

\( \text{seed} \). The difference of \( \text{tree-map-fold} \) and \( \text{tree-map-fold-right} \) is the associative order

of applying \( \text{proc} \), just like the difference between \( \text{fold} \) and \( \text{fold-right} \).

\begin{align*}
\text{tree-map-fold:} \\
& (\text{proc Kn Vn} (\text{proc Kn-1 Vn-1} \ldots (\text{proc K0 V0 seed})))
\end{align*}

\begin{align*}
\text{tree-map-fold-right:} \\
& (\text{proc K0 V0} (\text{proc K1 V1} \ldots (\text{proc Kn Vn seed})))
\end{align*}

Some examples:

\begin{align*}
& (\text{define tree} \ (\text{alist->tree-map} \ '((3 . a) (7 . b) (5 . c)) = \langle)))
\end{align*}

\begin{align*}
& (\text{tree-map-fold tree list* } '()) \\
\Rightarrow & (7 \ b \ c \ 3 \ a)
& (\text{tree-map-fold-right tree list* } '()) \\
\Rightarrow & (3 \ a \ c \ 7 \ b)
\end{align*}

\[ \text{tree-map-map} \quad \text{tree-map proc} \]  [Function]
Calls \( \text{proc} \), which must take two arguments, with each key/value pair in \( \text{tree-map} \), and collect

the results into a list and returns it. The order of results corresponds to the order of keys—

that is, the first element of the result list is what \( \text{proc} \) returns with minimum key and its

value, and the last element of the result list is what \( \text{proc} \) returns with the maximum key and

its value. (Note: Like \( \text{map} \), the order that \( \text{proc} \) is actually called is unspecified; \( \text{proc} \) is better
to be side-effect free.)

\[ \text{tree-map-for-each} \quad \text{tree-map proc} \]  [Function]
 Calls \( \text{proc} \), which must take two arguments, with each key/value pair in \( \text{tree-map} \), in the

increasing order of the keys. \( \text{proc} \) is called purely for side effects; the returned values are
discarded.

\[ \text{tree-map-floor} \quad \text{tree-map probe :optional fallback-key fallback-value} \]  [Function]
\[ \text{tree-map-ceiling} \quad \text{tree-map probe :optional fallback-key fallback-value} \]  [Function]
\[ \text{tree-map-predecessor} \quad \text{tree-map probe :optional fallback-key} \]  [Function]
\[ \text{tree-map-successor} \quad \text{tree-map probe :optional fallback-key fallback-value} \]  [Function]
These procedures search the entry which has the closest key to the given \( \text{probe} \). If such

an entry is found, returns two values, its key and its value. Otherwise, returns two values, \( \text{fallback-key} \) and \( \text{fallback-value} \), both defaulted to \#f.

The criteria of “closest” differ slightly among these procedures; \( \text{tree-map-floor} \) finds the
maximum key which is no greater than \( \text{probe} \); \( \text{tree-map-ceiling} \) finds the minimum key
which is no less than \( \text{probe} \); \( \text{tree-map-predecessor} \) finds the maximum key which is strictly
less than \( \text{probe} \); and \( \text{tree-map-successor} \) finds the minimum key which is strictly greater
than \( \text{probe} \).

\[ \text{tree-map-floor-key} \quad \text{tree-map probe optional fallback-key} \]  [Function]
\[ \text{tree-map-ceiling-key} \quad \text{tree-map probe optional fallback-key} \]  [Function]
tree-map-predecessor-key  

Like tree-map-floor etc., but only returns the key of the found entry (or fallback-key if there’s no entry which satisfies the criteria).

tree-map-successor-key  

Like tree-map-floor etc., but only returns the value of the found entry (or fallback-value if there’s no entry which satisfies the criteria).

tree-map-floor-value  

tree-map-ceiling-value  

tree-map-predecessor-value  

tree-map-successor-value  

Returns a generator (see Section 9.12 [Generators], page 436) that yields pairs of key and value such that the key is in the specified range. If the descending keyword argument is #f (default), it yields the pairs with increasing keys; otherwise, it yields them with descending keys.

The keyword argument > and >= specifies the lower bound of the key, including and excluding the given key value itself, respectively. If both are given, either one of them is considered.

The keyword argument < and <= specifies the upper bound of the key, including and excluding the given key value itself, respectively. If both are given, either one of them is considered.

\[
\text{(define tm (alist->tree-map '((0 . a) (1 . b) (2 . c) (3 . d) (4 . e))
\hspace{1cm} default-comparator))}
\]

\[
\text{(use gauche.generator)}
\]

\[
\text{(generator->list}
\hspace{1cm} (tree-map->generator/key-range tm :>= 1 :< 4))
\Rightarrow ((1 . b) (2 . c) (3 . d))
\]

\[
\text{(generator->list}
\hspace{1cm} (tree-map->generator/key-range tm :>= 1 :< 4 :descending #t))
\Rightarrow ((3 . d) (2 . c) (1 . b))
\]

\[
\text{(generator->list}
\hspace{1cm} (tree-map->generator/key-range tm :<= 2))
\Rightarrow ((0 . a) (1 . b) (2 . c))
\]

\[
\text{(generator->list}
\hspace{1cm} (tree-map->generator/key-range tm :> 2))
\Rightarrow ((3 . d) (4 . e))
\]

tree-map-keys  

Returns a list of all keys and all values, respectively. The keys and values are in ascending order of the keys.

tree-map-values  

Returns a list of pairs of keys and values for all entries. The pairs are in ascending order of the keys.
alist->tree-map alist :optional comparator [Function]
alist->tree-map alist key=? key<? [Function]

Creates a new tree map with the comparator or key=?/key<? procedures, then populates it with alist, each pair in which are interpreted as a cons of a key and its value. The meaning of comparator, key=? and key<? are the same as make-tree-map.

The following two procedures compares two tree maps with slightly different views.

tree-map-compare-as-sets tree-map1 tree-map2 :optional value=? fallback [Function]

Compares two tree maps as sets of entries. If we look at tree maps as sets of entries, we can define a partial order between two maps; they are equal to each other if they have exactly the same entries, and tree-map A is smaller than tree-map B if A us a strict subset of B.

If tree-map1 and tree-map2 are the same, 0 is returned. If tree-map1 is smaller than tree-map2, -1 is returned. If tree-map1 is greater than tree-map2, 1 is returned.

If one argument isn’t subset of the other, we can’t determine the order. In such a case, if fallback is given, it is returned. Otherwise, an error is signalled.

The comparators of tree-map1 and tree-map2 must be the same (equal?), otherwise an error is signalled. See Section 6.2.4 [Basic comparators], page 116, about the comparators.

An entry is equal to another entry if their keys match in terms of the comparator of the tree-map, and also their values match with the provided value=? predicate, which is defaulted to equal?.

```
(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (2 . b) (3 . c)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
⇒ 0

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
⇒ -1

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c) (4 . d) (2 . b)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
⇒ 1

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c) (4 . d)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator))
⇒ ERROR: tree-maps can’t be ordered

(tree-map-compare-as-sets
 (alist->tree-map '((1 . a) (3 . c) (4 . d)) default-comparator)
 (alist->tree-map '((3 . c) (1 . a) (2 . b)) default-comparator)
 eq?
 #f)
⇒ #f
```
tree-map-compare-as-sequences tree-map1 tree-map2 :optional value-cmp

[Function]

Comparates two tree maps as sequence of entries, ordered by keys. If both maps have entries
with the same key, we use a comparator value-cmp to break the tie (naturally, value-cmp
must have ordering predicate.) If value-cmp is omitted, default-comparator is used.

The comparators of tree-map1 and tree-map2 must be the same (equal?), otherwise an error
is signalled. See Section 6.2.4 [Basic comparators], page 116, about the comparators.

If tree-map1 and tree-map2 are the same, 0 is returned. If tree-map1 is smaller than tree-
map2, -1 is returned. If tree-map1 is greater than tree-map2, 1 is returned.

Unlike tree-map-compare-as-sets, this procedure defines total order of tree maps which
share the same comparator.

(tree-map-compare-as-sequences
  (alist->tree-map '((1 . a) (3 . c)) default-comparator)
  (alist->tree-map '((3 . c) (2 . b)) default-comparator))
⇒ -1

(tree-map-compare-as-sequences
  (alist->tree-map '((2 . b) (3 . d)) default-comparator)
  (alist->tree-map '((3 . c) (2 . b)) default-comparator))
⇒ 1

6.15 Procedures and continuations

In Scheme, procedures are fundamental blocks to build a program (see Section 4.3 [Making
procedures], page 47). A procedure represents a certain computation, possibly parameterized,
and can be applied to the actual arguments to execute the computation. Scheme also provides
the means to extract the continuation of the current computation and wraps it in a procedure
(see Section 6.15.7 [Continuations], page 226).

Gauche extends the concept of procedure application, allowing you to apply any object as
if it’s a procedure; for example, you can set up Gauche to accept ("abc" 2) as a valid
application syntax. See Section 6.15.6 [Applicable objects], page 225, for the details.

6.15.1 Procedure class and applicability

<procedure> [Builtin Class]

Represents a procedure. Ordinary Scheme procedures created by lambda is an instance of
this class, as well as built-in primitive procedures written in C. Note that, in Gauche, other
type of objects can behave as a procedure; so checking whether an object is a procedure or
not doesn’t mean much unless you want to mess around with Gauche internals.

procedure? obj [Function]

[R7RS base] Returns #t if obj is inherently applicable objects, #f otherwise. By inherently
applicable we mean Gauche unconditionally understands that obj can be called as a proce-
dure; an instance of <procedure> is so, as well as generic functions (<generic>) and methods
(<method>). See Section 7.4 [Generic function and method], page 349, for the details.

Since you can make any type of objects applicable at any time (see Section 6.15.6 [Applicable
objects], page 225), the fact that procedure? returned #f doesn’t mean that the object
cannot be applied. To check if an object can be applied or not, use applicable? below.

apply proc arg1 ... args [Function]

[R7RS base] Calls a procedure proc with a list of arguments, (arg1 . . . args). The last
argument args must be a proper list. Returns (a) value(s) proc returns.
Chapter 6: Core library

(apply list 'a 'b '(c d e)) ⇒ (a b c d e)

(apply + 1 2 '(3 4 5)) ⇒ 15

(applicable? obj class ...) [Function]
Checks if obj can be called with the types of arguments listed in class .... That is, when (applicable? foo <string> <integer>) returns #t, then you can call foo as (foo "x" -2), for example. (It doesn’t mean you won’t get an error; foo may be accept only nonnegative integers, which you cannot tell from the result of applicable?. But if applicable? returns #t, Gauche won’t complain “foo is not applicable” when you call foo.

This procedure takes applicable objects into account. So, for example, (applicable? #/a/ <string>) returns #t, for the regular expressions are applicable to a string (see Section 6.12 [Regular expressions], page 183).

For generic functions, applicable? returns #t if it has at least one method such that each of its specifiers is a superclass of the corresponding class argument given to applicable?.

(define-method foo ((x <sequence>) (y <integer>)) #f)

(APPPLICABLE? FOO <TOP> <TOP>) ⇒ #F

The second example returns #t since <string> is a subclass of <sequence>, while the third example returns #f since <hash-table> isn’t a subclass of <sequence>. The fourth example returns #f since <real> isn’t a subclass of <integer>.

Traditional Scheme procedures (such as ones created by lambda) only cares the number of arguments but not their types; it accepts any type as far as the number of arguments matches. To check such a condition, pass <top> as the argument class. (<top> is a superclass of all classes.)

(if (applicable? cons <top> <top>) ⇒ #T

If you want to check an object is applicable to a certain number of some class of arguments, you can pass <bottom> as the argument class instead. (<bottom> is a subclass of all classes.)

(define-method foo ((x <sequence>) (y <integer>)) #f)

(APPPLICABLE? FOO <TOP> <TOP>) ⇒ #F

The “type” returned by this procedure is just a hint. Currently, most Scheme-defined procedures doesn’t have much type info.

(procedure-type filter) ⇒ #<^ <top> <top> -> *>

Some of subrs (C-defined procedures) has a bit more info, derived from C type information.

(procedure-type cons) ⇒ #<^ <top> <top> -> <pair>>

Eventually we’ll improve this system to return more precise type info, and use it for optimizations and other static analysis.
6.15.2 Universal accessor

\(\sim\) \textit{obj key keys ...} \hspace{1cm} \textbf{[Function]}

\texttt{(setter \(\sim\)) \textit{obj key keys ...}} \hspace{1cm} \textbf{[Function]}

The procedure \(\sim\) can be used to access a part of various aggregate types.

\begin{itemize}
  \item Access to an element of a sequence by index
  \[
  (\sim 'a b c) 0 \Rightarrow a \\
  (\sim '#(a b c) 2) \Rightarrow c \\
  (\sim "abc" 1) \Rightarrow \#\b \\
  (\sim '#u8(10 20 30) 1) \Rightarrow 20
  \]
  \end{itemize}

\begin{itemize}
  \item Access to an element of a collection by key
  \[
  (\sim (hash-table 'eq? '(a . 1) '(b . 2)) 'a) \Rightarrow 1
  \]
  \end{itemize}

\begin{itemize}
  \item Access to a slot of an object by slot name
  \[
  (\sim (sys-localtime (sys-time)) 'hour) \Rightarrow 20
  \]
  \end{itemize}

The access can be chained:

\[
(\sim '((a b c) (d e f) (g h i)) 1 2) \Rightarrow f
\]

\[
(\sim (hash-table 'eq? '(a . "abc") '(d . "def")) 'a 2) \Rightarrow \#\c
\]

You can think \(\sim\) as left-associative, that is,

\[
(\sim x k j) \equiv (\sim (\sim x k) j)
\]

and so on.

The generalized setter \texttt{set!} can be used with \(\sim\) to replace the specified element.

\begin{itemize}
  \item (define z (vector 'a 'b 'c))
  \item (set! (\sim z 1) 'Z)
  \item \(z \Rightarrow \#(a Z c)\)
  \end{itemize}

\begin{itemize}
  \item (define z (vector (list (vector 'a 'b 'c) (vector 'd 'e 'f) (vector 'g 'h 'i)) (list (vector 'a 'b 'c) (vector 'd 'e 'f) (vector 'g 'h 'i))))
  \item \(z \Rightarrow \#((#(a b c) #(d e f) #(g h i)) #(a b c) #(d e f) #(g h i)))\)
  \end{itemize}

\begin{itemize}
  \item (set! (\sim z 1 2 0) 'Z)
  \item \(z \Rightarrow \#((#(a b c) #(d e f) #(g h i)) #(a b c) #(d e f) #(Z h i)))\)
  \end{itemize}

Internally, a call to \(\sim\) is implemented by a generic function \texttt{ref}. See Chapter 7 [Object system], page 329, for more about generic functions.
Many aggregate types defines a specialized method of these to provide uniform access and
mutation. Meaning of optional arguments args of ref depends on each specialized method,
but it is common that the first optional argument of ref is a fallback value, which is to be
returned when object doesn’t have a meaningful association with key.

The manual entry of each aggregate type shows the specialized method and its semantics in
detail.

Conceptually, ~ can be understood as follows:

```
(define ~
  (getter-with-setter
    (case-lambda
      [[(obj selector) (ref obj selector)]
        [[(obj selector . more) (apply ~ (ref obj selector) more))]
      (case-lambda
        [[(obj selector val) ((setter ref) obj selector val)]
          [[(obj selector selector2 . rest)
             (apply (setter ~) (ref obj selector) selector2 rest))))]))
```

(Gauche may use some short-cut for optimization, though, so this code may not reflect the
actual implementation.)

### 6.15.3 Combinators
Gauche has some primitive procedures that allows combinatory programming.

```
pa$ proc arg ... [Function]
Partial application. Returns a procedure, and when it is called with arguments m . . ., it is
equivalent to call (proc arg ... m ...).

(define add3 (pa$ + 3))
(add3 4) ⇒ 7
```

```
(map (pa$ * 2) ’(1 2 3)) ⇒ (2 4 6)
```

Macros cut and cute defined in SRFI-26 provide a similar abstraction, with a bit more
flexible but less compact notation. See Section 4.3 [Making procedures], page 47.

SRFI-235 defines left-section, which is effectively the same as pa$ (see Section 11.60
[Combinators (SRFI)], page 804).

```
apply$ proc [Function]
map$ proc [Function]
for-each$ proc [Function]
```

Partial application versions of apply, map and for-each.

```
(define map2* (map$ (pa$ * 2)))
(map2* ’(1 2 3)) ⇒ (2 4 6)
```

```
count$ pred [Function]
fold$ kons :optional knil [Function]
fold-right$ kons :optional knil [Function]
reduce$ f :optional ridentity [Function]
reduce-right$ f :optional ridentity [Function]
filter$ pred [Function]
remove$ pred [Function]
```
Chapter 6: Core library

partition$ pred
member$ item
find$ pred
find-tail$ pred
any$ pred
every$ pred
delete$ pred
assoc$ item

Partial application versions of some SRFI-1 (R7RS (scheme list)) procedures (see Section 10.3.1 [R7RS lists], page 593).

$. f ... [Function]
compose f ... [Function]

Combine procedures. All arguments must be procedures. When two procedures are given, $(. f g)$ is equivalent to the following code:

$\lambda$ args (call-with-values (lambda () (apply g args)) f)

When more than two arguments are passed, they are composed as follows:

$(. f g h ... ) \equiv (.(. f g) h ... )$

Some examples:

(define not-zero? (.= not zero?))
(not-zero? 3) ⇒ #t
(not-zero? 0) ⇒ #f

(define dot-product (.= (apply$ +) (map$ *)))
(dot-product '(1 2 3) '(4 5 6)) ⇒ 32

A couple of edge cases: if only one argument is given, the argument itself is returned. If no arguments are given, the procedure values is returned.

Note: The name $. comes from the fact that . is commonly used for function composition in literatures and some programming languages, and that Gauche uses suffix $ to indicate combinators. However, since it is not a valid R7RS identifier, portable programs may want to use the alias compose, with which you can easily add a portable definition using SRFI-0, for example.

SRFI-210 provides compose-left and compose-right; see Section 11.49 [Procedures and syntax for multiple values], page 784.

identity obj [Function]

Returns obj. You can use values to get the same effect for the porable code, but using this emphasizes that you’re dealing with exactly one value.

Note: SRFI-210 introduces multi-valued identity (unfortunately), which is a synonym to values (see Section 11.49 [Procedures and syntax for multiple values], page 784).

constantly obj ... [Function]

[SRFI-235] Returns a procedure that takes arbitrary number of arguments, discards them, and returns obj obj .... Same as (^ _ (values obj ...)).

complement pred [Function]

[SRFI-235] Returns a procedure that reverses the meaning of the predicate pred. That is, for the arguments for which pred returns true return false, and vice versa.

(map (complement even?) '(1 2 3)) ⇒ '(#t #f #t)
(map (complement =) '(1 2 3) '(1 1 3)) ⇒ '(#f #t #f)
(((complement (lambda () #f))) ⇒ #t
flip proc
[SRFI-235] Returns a procedure that passes its arguments to proc in reverse order, that is, 
((flip proc) a b ... y z) is (proc z y ... b a).

swap proc
[SRFI-235] Proc must be a procedure that takes two arguments or more. Returns a procedure 
that passes its first and second arguments swapped to proc, that is, 
((swap proc) a b arg ...) is (proc b a arg ...).

any-pred pred...
Returns a procedure which applies given argument(s) to each predicate pred. If any pred 
returns a non-#f value, the value is returned. If all the preds return #f, #f is returned.

```
(define string-or-symbol? (any-pred string? symbol?))
(string-or-symbol? "abc") ⇒ #t
(string-or-symbol? 'abc) ⇒ #t
(string-or-symbol? 3) ⇒ #f
```

```
(define <> (any-pred < >))
(< 3 4) ⇒ #t
(< 3 3) ⇒ #f
```

```
((any-pred (cut memq <> '(a b c))
            (cut memq <> '(1 2 3)))
   'b) ⇒ '(b c)
```

every-pred pred...
Returns a procedure which applies given argument(s) to each predicate pred. If every pred 
returns a non-#f value, the value returned by the last pred is returned. If any pred returns #f, every-pred returns #f without calling further preds.

```
(every-pred odd? positive?) 3) ⇒ #t
(every-pred odd? positive?) 4) ⇒ #f
((every-pred odd? positive?) -3) ⇒ #f
```

```
(define safe-length (every-pred list? length))
(safe-length '(a b c)) ⇒ 3
(safe-length "aaa") ⇒ #f
```

6.15.4 Optional argument parsing
Gauche supports optional and keyword arguments in extended lambda syntax (see Section 4.3 [Making procedures], page 47). However, you can also use the following macros to parse optional and keyword arguments, without relying Gauche’s extension.

```
(define (foo a b :optional (c #f) (d 'none))
  body ...)
```

;; is roughly equivalent to ...

```
(define (foo a b . args)
  (let-optionals* args ((c #f) (d 'none))
    body ...))
```

Explicitly parsing the extended arguments may be useful for portable programs, since it is rather straightforward to implement those macros rather than extend lambda syntax.

Those macros can also be useful to factor out common argument parsing routines.
let-optionals* restargs (var-spec . . .) body . . .  [Macro]
let-optionals* restargs (var-spec . . . . . restvar) body . . .  [Macro]

Given a list of values restargs, binds variables according to var-spec, then evaluates body.

Var-spec can be either a symbol, or a list of two elements and its car is a symbol. The symbol is the bound variable name. The values in restargs are bound to the symbol in order. If there are not as many values in restargs as var-spec, the rest of symbols are bound to the default values, determined as follows: If var-spec is just a symbol, the default value is undefined. If var-spec is a list, the default value is the result of evaluation of the second element of the list. In the latter case the second element is only evaluated when there are not enough arguments. The binding proceeds in the order of var-spec, so the second element may refer to the bindings of previous var-spec.

In the second form, restvar must be a symbol and bound to the list of values whatever left from restargs after binding to var-spec.

It is not an error if restarg has more values than var-specs. The extra values are simply ignored in the first form.

```
(define (proc x . args)
  (let-optionals* args ((a 'a) (b 'b) (c 'c))
    (list x a b c)))
```

```
(proc 0)  ⇒ (0 a b c)
(proc 0 1) ⇒ (0 1 b c)
(proc 0 1 2) ⇒ (0 1 2 c)
(proc 0 1 2 3) ⇒ (0 1 2 3)
```

```
(define (proc2 . args)
  (let-optionals* args ((a 'a) . b)
    (list a b)))
```

```
(proc2)  ⇒ (a ())
(proc2 0) ⇒ (0 ())
(proc2 0 1) ⇒ (0 (1))
(proc2 0 1 2) ⇒ (0 (1 2))
```

```
(define (proc3 . args)
  (let-optionals* args ((a 0)
                        (b (+ a 1))
                        (c (+ b 1)))
    (list a b c)))
```

```
(proc3)  ⇒ (0 1 2)
(proc3 8) ⇒ (8 9 10)
(proc3 8 2) ⇒ (8 2 3)
(proc3 8 2 -1) ⇒ (8 2 -1)
```

get-optional restargs default  [Macro]

This is a short version of let-optionals* where you have only one optional argument. Given the optional argument list restargs, this macro returns the value of optional argument if one is given, or the result of default otherwise. Default is not evaluated unless restargs is an empty list.
(define (proc x . maybe-opt)
  (let ((option (get-optional maybe-opt #f)))
    (list x option)))

(proc 0)  ⇒  (0 #f)
(proc 0 1) ⇒  (0 1)

(let-keywords restarg (var-spec ...) body ...)  [Macro]
(let-keywords restarg (var-spec ... . restvar) body ...)  [Macro]

This macro is for keyword arguments. Var-spec can be one of the following forms:

(symbol expr)
If the restarg contains keyword which has the same name as symbol, binds symbol to the corresponding value. If such a keyword doesn’t appear in restarg, binds symbol to the result of expr.

(symbol keyword expr)
If the restarg contains keyword keyword, binds symbol to the corresponding value. If such a keyword doesn’t appear in restarg, binds symbol to the result of expr.

The default value expr is only evaluated when the keyword is not given to the restarg.
If you use the first form, let-keyword throws an error when restarg contains a keyword argument that is not listed in var-specs. When you want to allow keyword arguments other than listed in var-specs, use the second form.
In the second form, restvar must be either a symbol or #f. If it is a symbol, it is bound to a list of keyword arguments that are not processed by var-specs. If it is #f, such keyword arguments are just ignored.

(define (proc x . options)
  (let-keywords options ((a 'a)
    (b :beta 'b)
    (c 'c)
    . rest)
    (list x a b c rest)))

(proc 0)  ⇒  (0 a b c ())
(proc 0 :a 1) ⇒  (0 1 b c ())
(proc 0 :beta 1) ⇒  (0 a 1 c ())
(proc 0 :beta 1 :c 3 :unknown 4) ⇒  (0 a 1 3 (:unknown 4))

(let-keywords* restarg (var-spec ...) body ...)  [Macro]
(let-keywords* restarg (var-spec ... . restvar) body ...)  [Macro]

Like let-keywords, but the binding is done in the order of var-specs. So each expr can refer to the variables bound by preceding var-specs.

6.15.5 Procedure arity

Interface to query procedure’s arity. The API is taken from MzScheme (PLT Scheme).

arity proc  [Function]
Given procedure proc, returns an integer, an arity-at-least object, or a list of integer(s) and arity-at-least objects.

An integer result indicates proc takes exactly that number of arguments. An arity-at-least indicates proc takes at least (arity-at-least-value arity-at-least) arguments. The list indicates there are multiple procedures with different arities.
Since one can add methods to an existing procedure or generic function at any moment in Gauche, the value returned by `arity` only indicates the current state of the procedure. It will change if new method is added to the procedure/generic-function.

```scheme
(arity cons) ⇒ 2
(arity list) ⇒ #<arity-at-least 0>
(arity make) ⇒ (#<arity-at-least 1>)
```

`arity-at-least? obj`  
Returns true if `obj` is an arity-at-least object.

`arity-at-least-value arity-at-least`  
Returns the number of required arguments the arity-at-least object indicates.

`procedure-arity-includes? proc k`  
If a procedure `proc` can take `k` arguments, returns `#t`. Otherwise returns `#f`.

### 6.15.6 Applicable objects

Gauche has a special hook to make an arbitrary object applicable.

```scheme
object-apply object arg ...  
```

If an object that is neither a procedure nor a generic function is applied to some arguments, the object and the arguments are passed to a generic function `object-apply`.

This can be explained better by examples.

For example, suppose you try to evaluate the following expression:

```scheme
("abcde" 2)
```

The operator evaluates to a string, which is neither a procedure nor a generic function. So Gauche interprets the expression as if it were like this:

```scheme
(object-apply "abcde" 2)
```

Gauche doesn’t define a method of `object-apply` that takes `<string>` and `<integer>` by default, so this signals an error. However, if you define such a method:

```scheme
(define-method object-apply ((s <string>) (i <integer>))
  (string-ref s i))
```

Then the first expression works as if a string is applied on the integer:

```scheme
("abcde" 2) ⇒ #\c
```

This mechanism works on almost all occasions where a procedure is allowed.

```scheme
(apply "abcde" '(1)) ⇒ (#\b)
(map "abcde" '(3 2 1)) ⇒ (#\d #\c #\b)
```

Among Gauche built-in objects, `<regexp>` object and `<regmatch>` object have `object-apply` defined. See Section 6.12 [Regular expressions], page 183.

Note: Defining `object-apply` method affect globally. Although it is cool to define `object-apply` on strings as in the above example, it is not recommended as a general trick, for it’ll change the behavior of string across the entire program.

Keep the above example personal experiments. In general, you should only define `object-apply` on your own classes.

```scheme
(setter object-apply) object arg ... value
```

If a form of applying an applicable object appears in the first position of `set!` form, this method is called, that is:

```scheme
(set! (object arg ...) value)
⇒ ((setter object-apply) object arg ... value)
```
6.15.7 Continuations

\texttt{call-with-current-continuation proc} \hspace{1em} [Function]

\texttt{call/cc proc} \hspace{1em} [Function]

[R7RS base] Encapsulates the current continuation to a procedure (“continuation procedure”), and calls \texttt{proc} with it. When \texttt{proc} returns, its value becomes \texttt{call/cc}’s value. When the continuation procedure is invoked with zero or more arguments somewhere, the further calculation is abandoned and \texttt{call/cc} returns with the arguments given to the continuation procedure.

First class continuation is one of the most distinct feature of Scheme, but this margin is too small to contain explanation. Please consult to the appropriate documents.

There’s a nontrivial interaction between C language runtime and Scheme continuation. Suppose the following scenario:

1. An application’s C runtime calls back a Scheme routine. For example, GUI framework calls back a draw routine written in Scheme.
2. A continuation is captured in the Scheme routine.
3. The Scheme routine returns to the C runtime.
4. The continuation captured in 2 is invoked.

It is no problem to invoke the continuation, but if the control is about to return to the Scheme routine to the C runtime (that is, to execute step 3 again), an error is signaled as follows.

\texttt{*** ERROR: attempt to return from a ghost continuation.}

This is because C routines don’t expect the calling function to return more than once. The C stack frame on which the Scheme callback was originally called is likely to be deallocated or modified at the time the continuation is invoked.

If you think of a continuation as a chain of control frames, growing from root towards upward, you can imagine that, once a control returns to the C world, the chain is cut at the boundary. You can still execute such rootless continuations, but you have to move the control away from it before it tries to return to its root that no longer exists. You can call another continuation, or raise an exception, for example.

Using partial continuations (or delimited continuations) is another way to avoid such complications. See Section 9.26 [Partial continuations], page 485.

\texttt{let/cc \var body ...} \hspace{1em} [Macro]

This macro expands to : (\texttt{call/cc (lambda (var) body ...)}). The API is taken from PLT Scheme.

\texttt{dynamic-wind before body after} \hspace{1em} [Function]

[R7RS base] This is a primitive to manage \texttt{dynamic environment}. Dynamic environment is a set of states which are kept during execution of a certain expression. For example, the current output ports are switched \texttt{during} execution of \texttt{with-output-to-port}. They can be nested dynamically, as opposed to the lexical environment, in which nesting is determined statically from the program source.

Before, \texttt{body} and \texttt{after} are all procedures with no arguments. In normal situation, \texttt{dynamic-wind} calls \texttt{before}, then \texttt{body}, then \texttt{after}, then returns whatever value(s) \texttt{body} returned.

The intention is that the \texttt{before} thunk sets up the dynamic environment for execution of \texttt{body}, and the \texttt{after} thunk restores it to the previous state.

If a control flow goes out from \texttt{body} by invoking a continuation captured outside of the dynamic scope of \texttt{dynamic-wind} (for example, an error is signaled in \texttt{body}), \texttt{after} is called.
If a control flow goes into body by invoking a continuation captured inside body from outside of the dynamic scope of \texttt{dynamic-wind}, before is called.

\begin{verbatim}
(letrec ((paths '())
  (c #f)
  (add (lambda (s) (push! paths s))))
(dynamic-wind
  (lambda () (add 'connect))
  (lambda ()
    (add (call/cc (lambda (c0) (set! c c0) 'talk1))))
  (lambda () (add 'disconnect)))
(if (< (length paths) 4)
  (c 'talk2)
  (reverse paths)))
⇒ (connect talk1 disconnect connect talk2 disconnect)
\end{verbatim}

Note: Since \texttt{after} is guaranteed to be called when an error causes body to abort, it may appear tempting to use \texttt{dynamic-wind} to use resource clean-up, just like \texttt{try-catch} construct in Java. It’s \textit{not} for that. Since the control may return to body, the situation \texttt{dynamic-wind} handles should be considered more like a context switch.

For resource clean-up, you can use exception handling mechanism such as \texttt{guard} and \texttt{unwind-protect} (see Section 6.19.3.1 [High-level exception handling mechanism], page 249), which is built on top of \texttt{dynamic-wind}.

As a rule of thumb, \texttt{after} should do things that can be reverted by \texttt{before}, such as manipulating error handler stack (instead of actually handling errors).

### 6.15.8 Continuation prompts

Continuations are often explained as “the rest of computation”. It is enough for theorists to write papers, but in practice, we cannot reify the computations “from now on for ever”. There has to be an end—b it may be until a single program terminates, or until a server returns a response to a request, or a REPL prints the result of evaluation. Implementations have chosen the “end” of continuations suitable for their needs. Sometimes, though, the ambiguity caused confusion.

Continuation prompts are to make the “end” of continuation explicit. It is called a prompt because, in the interactive environment, REPL prompt is usually delimits the continuation. If you think of activation records as a stack, a continuation prompt is the “bottom” of the stack.

Furthermore, you can have multiple kinds of continuation prompts, so that multiple features that use continuation prompts independently won’t step on each other.

(Partial continuations can also be defined on top of continuation prompts, and SRFI-226 indeed defines so. We’ll integrate existing partial continuations with continuation prompts in future versions.)

\begin{verbatim}
make-continuation-prompt-tag :optional name
\end{verbatim} 

[Function] \cite{SRFI-226} Creates and returns a new continuation prompt tag, which is an opaque object to specify the boundary of continuation frames. A new boundary can be introduced with \texttt{call-with-continuation-prompt}. You can nest bounded continuation frames with the same continuation prompt tag. Operations that takes a continuation prompt tag seraches the innermost boundary with the tag.

The optional \texttt{name} argument is the name of the continuation prompt tag. It is for display purpose only to help diagnostics.
default-continuation-prompt-tag
   [Function]
   [SRFI-226] Returns a default continuation prompt tag. When the program starts, the initial
   continuation boundary has the default continuation prompt tag. Most procedures that takes
   an optional continuation prompt tag assumes the default tag if the argument is omitted.

continuation-prompt-tag? obj
   [Function]
   [SRFI-226] Returns #t if obj is a continuation prompt tag, #f otherwise.

call-with-continuation-prompt thunk :optional prompt-tag abort-handler
   [Function]
   [SRFI-226] Calls thunk with a continuation delimited by a continuation prompt tag prompt-
   tag. If prompt-tag is omitted, the default prompt tag is assumed.
   The abort-handler is a procedure to be called when, during executing thunk, its continua-
   tion is aborted by abort-current-continuation. The objects passed to abort-current-
   continuation are passed to abort-handler, which is called with the continuation of call-with-
   continuation-prompt itself. If thunk doesn’t abort the continuation, abort-handler won’t be
   called.
   When abort-handler is omitted, it installs a default abort handler, which work in a very
   specific way. First, it takes one argument, which must be a thunk (let’s call it continuation-
   thunk for now). The default abort handler reinstates the continuation with the continuation
   prompt tag prompt-tag and calls the continuation thunk. Say, it works like a trampoline to
   push back the control to the same context as thunk is called.
   See the entry of abort-current-continuation below for examples.

abort-current-continuation prompt-tag obj ...
   [Function]
   [SRFI-226] Aborts all active procedure calls in the current continuation up to the continuation
   prompt tagged with prompt-tag. The control goes to the abort handler installed by the most
   recent call-with-continuation-prompt with the prompt-tag. The rest of the arguments
   obj ... are passed to the abort handler.
   When the current continuation does not have a continuation prompt tagged with prompt-tag,
   a <continuation-violation> error is thrown.

   (let ([tag (make-continuation-prompt-tag)])
     (call-with-continuation-prompt
      (^
       (call-with-continuation-prompt
        (^
         (+ 1 (abort-current-continuation tag 'foo 'bar) 2))
        tag
        list))
      tag
      list))
   ⇒ (foo bar)

   If a dynamic handlers are inserted in the continuation frames to be aborted, their ‘after’
   handlers are executed before the abort handler is called.

   (let ((tag (make-continuation-prompt-tag)))
     (call-with-continuation-prompt
      (^
       (dynamic-wind
        (^
         #f)
        (^
         (abort-current-continuation tag 'foo))
        (^
         (print 'yo)))
       tag
       print))
   ⇒ prints yo, then prints foo.

   You may wonder the difference between the traditional call/cc and the combination of
   call-with-continuation-prompt and abort-current-continuation. Actually the latter is
   more lower-level construct, and call/cc is a high-level construct implemented on top of it.
6.15.9 Continuation marks

Continuation marks are a set of key-value associations attached to continuation frames. Keys and values are arbitrary Scheme objects. It sounds obscure, but it allows to implement various useful language features; notably, the new parameterize is implemented on top of continuation marks, allowing its body to be evaluated in tail context (see Section 6.16.1 [Parameters], page 233).

- You can have dynamic-scoped data without compromising tail-calls. Using dynamic-wind requires to run the ‘after’ thunk after evaluating the body, so if you wrap a tail expression with it, the original expression won’t be in tail context anymore. Continuation marks, on the other hand, will be discarded when the continuation frame they are attached to is ‘popped’, so you don’t need to clean up after the body is executed. So the body is executed in the tail context if the dynamic-scope itself is in tail context.
- Since tail expression shares a continuation frame with its caller, you can detect whether an expression is evaluated at the tail position at run-time.

In general, though, continuations marks are low-level tools to construct high-level features, rather than to be used as day-to-day tools.

Marking a continuation frame

with-continuation-mark key value expr [Special Form]
[SRFI-226] Both key and value are evaluated, and adds the key-value association as a mark to the current continuation, then evaluate expr with the same continuation as this form. That is, if this form is in the tail context, expr is evaluated in the same tail context. If the current continuation is already marked with key, then its value is replaced with value.

with-continuation-mark ((key value) . . .) expr [Macro]
[SRFI-226] All keys and values are evaluated, then add those key-value associations as marks to the current continuation, then evaluate expr with the same continuation as this form.

Extracting continuation marks

call-with-immediate-continuation-mark key proc :optional fallback [Function]
[SRFI-226] Tail-calls proc with the value for key associated in the current continuation. If the current continuation doesn’t have a value for key, fallback is passed to proc instead. If fallback is omitted, #f is assumed.

(with-continuation-mark 'a 'b
  (call-with-immediate-continuation-mark 'a list))
⇒ (b)

(with-continuation-mark 'a 'b
  (cons (call-with-immediate-continuation-mark 'a list) 1))
⇒ ((#f) . 1)

continuation-marks cont :optional prompt-tag [Function]
[SRFI-226] Returns a continuation mark set object, which contains continuation marks of the continuation frames between the continuation cont and the continuation delimited by prompt-tag. If prompt-tag is omitted, default prompt tag is assumed.

current-continuation-marks :optional prompt-tag [Function]
[SRFI-226] Returns a continuation mark set object, which contains continuation marks of the continuation frames between the current continuation and the continuation delimited by prompt-tag. If prompt-tag is omitted, default prompt tag is assumed.
Continuation mark set

continuation-mark-set? obj

[Function]  [SRFI-226] Returns #t if obj is a continuation mark set, #f otherwise.

continuation-mark-set->list mark-set key :optional prompt-tag

[Function]  [SRFI-226] Pick up values associated to key from a continuation mark set mark-set, and returns a fresh list of them. A value in more recent continuation frames come earlier in the list. You can pass #f to mark-set to make the procedure use (current-continuation-marks prompt-tag).

It searches the mark set up to the continuation frame delimited by a continuation prompt prompt-tag. If prompt-tag is omitted, the default prompt tag is used.

continuation-mark-set-first mark-set key :optional fallback prompt-tag

[Function]  [SRFI-226] Returns the most recent value associated with key in the continuation mark set mark-set. If there’s a mark with key exists, it matches (car (continuation-mark-set->list key prompt-tag)).

If there’s no mark with key in the mark set, fallback is returned. If fallback is omitted, #f is assumed.

If prompt-tag is omitted, the default prompt tag is used.

Continuation mark key

You can use arbitrary Scheme object as keys for continuation marks, but you if you’re writing libraries, you want to make sure that your keys won’t conflict with the keys used by other parts of the code.

A continuation mark key is an opaque object that is guaranteed to be different (in terms of equal?) to other continuation mark keys.

make-continuation-mark-key :optional name

[Function]  [SRFI-226] Returns a newly created continuation key, which is guaranteed never to be equal to any prior or future continuation mark keys.

The optional name argument is for debugging purpose.

Note that SRFI-226 does not require a continuation mark key has disjoint types from other Scheme types. Currently, Gauche uses an uninterned symbol as a continuation mark key. It may be changed in future, so do not count on it.

continuation-mark-key? obj

[Function]  [SRFI-226] Returns #t if obj can be an object returned by make-continuation-mark-key, #f otherwise.

Note that SRFI-226 does not require a continuation mark key has a disjoint type, so this may return #t for an object that’s not created by make-continuation-mark-key. In fact, Gauche currently uses an uninterned symbol for a continuation mark key, and this procedure returns #t for any symbols.

6.15.10 Multiple values

values obj ...  

[Function]  [R7RS base] Returns obj ... as multiple values. Caller can capture multiple values by a built-in syntax receive or let-values (Section 4.6 [Binding constructs], page 59), or the R7RS procedure call-with-values described below.

(values 1 2) ⇒ 1 and 2
call-with-values  producer  consumer  

[Function]  
[R7RS base] Call a procedure producer with no argument. Then applies a procedure consumer on the value(s) producer returned. Returns the value(s) consumer returns.  

\[
(call\text{-}with\text{-}values\ (\lambda ()\ (values\ 1\ 2))\ cons) \\
\Rightarrow\ (1\ .\ 2)
\]

values-ref  mv-expr  k  

[Macro]  
Returns k-th value of what mv-expr returns. Conceptually, it is the same as the following code.  

\[
(call\text{-}with\text{-}values\ (\lambda ()\ mv\text{-}expr)\ (\lambda r\ (list\text{-}ref\ r\ k)))
\]

This macro uses shortcuts for the typical cases like k is zero.  
Similar to Common Lisp's nth-value, but the argument order is flipped to match other Scheme's *-ref procedures.

values->list  mv-expr  

[Macro]  
Evaluates mv-expr, puts all the results into a list and returns it. It is called multiple-value-list in Common Lisp.  

\[
(values\text{-}to\text{-}list\ (div\text{-}and\text{-}mod\ 10\ 3))\Rightarrow\ (3\ 1)
\]

\[
(values\text{-}to\text{-}list\ 1)\Rightarrow\ (1)
\]

6.15.11 Folding generated values  

Sometimes a procedure is used as a generator of a series of values, by yielding one value at a time. Customary an EOF object is used to mark the end of the series. For example, read-char is such a procedure that yields a series of characters, terminated by EOF.  

Since it is such a handy abstraction, Gauche provides a set of utilities (see Section 9.12 [Generators], page 436) to construct and generators out of various sources, including other generators.  

The generated values needs to be consumed eventually. Here we provide several procedures to do that. These are useful when combined with input procedures like read, so we have them built-in instead of putting them in a separate module.

generator-fold  proc  seed  gen  gen2  . . .  

[Function]  
[R7RS generator] Works like fold on the generated values by generator procedures gen gen2 . . . (See Section 6.6.6 [Walking over lists], page 147, for the details of fold).  

When one generator is given, for each value v generated by gen, proc is called as \( (proc\ v\ r) \), where r is the current accumulated result; the initial value of the accumulated result is seed, and the return value from proc becomes the next accumulated result. When gen returns EOF, the accumulated result at that time is returned from generator-fold.  
When more than one generator is given, proc is called as \( (proc\ v1\ v2\ . . .\ r) \), where v1, v2 . . . are the values yielded from gen, gen2, . . ., respectively, and r is the current accumulated result. The iteration terminates when any one of the generators returns EOF.  

\[
(with\text{-}input\text{-}from\text{-}string\ "a\ b\ c\ d\ e"
(cut\\ generator\text{-}fold\ cons\ 'z\ read))
\Rightarrow\ (e\ d\ c\ b\ a\ .\ z)
\]

generator-fold-right  proc  seed  gen  gen2  . . .  

[Function]  
Works like fold-right on the generated values by generator procedures gen gen2 . . . (see Section 6.6.6 [Walking over lists], page 147, for the details of fold-right).
This is provided for completeness, but it isn’t a good way to handle generators; in order to combine values right-associatively, we should read all the values from the generators (until any one of the generator returns EOF), then start calling proc as

\[
\text{proc } v_{0,0} \text{ } v_{1,0} \ldots \text{ (proc } v_{0,1} \text{ } v_{1,1} \ldots \text{ (proc } v_{0,n} \text{ } v_{1,n} \ldots \text{ seed} \ldots)\]
\]

where \(v_{n,m}\) is the \(m\)-th value yielded by \(n\)-th generator.

\[
\text{(with-input-from-string "a b c d e"}
\]

\[
\text{ (cut generator-fold-right cons 'z read))}
\]

\[
\Rightarrow (a \text{ } b \text{ } c \text{ } d \text{ } e \text{ } . \text{ } z)
\]

As you see, keeping all intermediate values kind of defeats the benefit of generators.

**generator-for-each proc gen gen2 ...** [Function]

[R7RS generator] A generator version of `for-each`. Repeatedly applies proc on the values yielded by gen, gen2 ... until any one of the generators yields EOF. The values returned from proc are discarded.

This is a handy procedure to consume generated values with side effects.

**generator-map proc gen gen2 ...** [Function]

A generator version of map. Repeatedly applies proc on the values yielded by gen, gen2 ... until any one of the generators yields EOF. The values returned from proc are collected into a list and returned.

\[
\text{(with-input-from-string "a b c d e"}
\]

\[
\text{ (cut generator-map symbol->string read))}
\]

\[
\Rightarrow ("a" \text{ } "b" \text{ } "c" \text{ } "d" \text{ } "e")
\]

The same effects can be achieved by combining generator->list and gmap (see Section 9.12.2 [Generator operations], page 440). This procedure is provided for the backward compatibility.

\[
\text{(generator->list (gmap proc gen gen2 ...))}
\]

**generator-find pred gen** [Function]

[R7RS generator] Returns the first item from the generator gen that satisfies the predicate pred.

The following example returns the first line matching the regexp #/XYZ/ from the file foo.txt.

\[
\text{(with-input-from-file "foo.txt"}
\]

\[
\text{ (cut generator-find #/XYZ/ read-line))}
\]

Note: If you want to pick all the lines matching the regexp, like the grep command, you can use gfilter and generator->list.

### 6.16 Parameters and dynamic states

A parameter is Scheme’s way to implement dynamically scoped variables. It is codified in SRFI-39 and incorporated into R7RS.

**Note:** Starting from 0.9.13, we’re adopting SRFI-226 semantics of parameters, which behaves slightly different from our legacy parameters. See Section 6.16.4 [Transition to SRFI-226 parameters], page 237, for the details.

It is a special procedure created by make-parameter. It accepts zero or one argument. When called without an argument, it returns the value in the current dynamic environment. If an argument is passed, it alters its value with the given value in the current dynamic environment, and returns the previous value.

Using parameterize macro, which is similar to let syntactically, rebinds the parameters’ value during execution of its body, dynamically rather than lexically.

\[
\text{(define var (make-parameter 0))}
\]
(define (show-var) (print (var)))

(show-var) ; prints 0

(parameterize ((var 1))
  (show-var)) ; prints 1

(define f)

(parameterize ((var 2))
  (set! f (lambda () (show-var))))

(f) ; prints 0, since its out of the dynamic extent of var=2

A parameter’s value can be mutated by calling the parameter with the new value:

(define var (make-parameter 0))

(var) ;⇒ 0

(var 1)

(var) ;⇒ 1

;; Gauche extension: you can use set! too
(set! (var) 2)

(var) ;⇒ 2

Originally, parameters in Scheme were simply a hack with an ordinary procedure and \texttt{dynamic-wind} to emulate dynamically scoped variables. The same mechanism have been used to manage dynamically scoped states, too.

As the experience accumulated, however, it is learned that it’s better to treat dynamically scoped variables and dynamically scoped states differently; efficient implementations differ, and they have slightly different semantics with delimited continuations. SRFI-226 incorporated this distinction.

As of Gauche 0.9.13, we adopt this distinction, too. See Section 6.16.3 [Difference of parameters and dynamic states], page 236, for the details.

If you’re a long time Gauche user (thanks!), remember to check out the changes on 0.9.13 and how to transition to it (see Section 6.16.4 [Transition to SRFI-226 parameters], page 237).

### 6.16.1 Parameters

Because of historical reasons, parameters come with three flavors; their differences are subtle, and only visible when you use \textit{mutation} in \textit{multi-thread code}. If you only use parameters for dynamic binding (i.e. within R7RS spec), all flavors of parameters behave the same.

The three flavors are \textit{shared parameters}, \textit{thread parameters}, and \textit{legacy parameters}. The first two is defined in SRFI-226 (though it doesn’t use the term “shared”, for the ordinary parameters are shared parameters). The third one is compatible to the parameters in Gauche 0.9.12 and before.

The following table shows whether a mutation in one thread is visible from another thread:
Toplevel mutation visible? yes no no
Dynamic mutation visible? yes no yes

Here, “toplevel mutation” is a mutation done for a parameter outside of any parameterize form, and “dynamic mutation” is a mutation done for a parameter in a dynamic scope of parameterize form:

```
(define p (make-parameter 0))

(p 1) ; toplevel mutation

(parameteize ((p 2))
  (p 3) ; dynamic mutation
)

(p 4) ; toplevel mutation
```

For the shared and legacy parameters, dynamic mutation is only visible for the thread that shares the same dynamic binding.

```
make-parameter value :optional converter
make-shared-parameter value :optional converter
make-thread-parameter value :optional converter
make-legacy-parameter value :optional converter
```

[R7RS base][SRFI-226] Creates a parameter whose initial value is value. If an optional argument converter is given, it must be a procedure that takes one argument and returns one value; whenever the parameter’s value is about to change, the procedure is called with the given value, and the value the procedure returns will be the parameter’s value. The converter procedure can raise an error or reject to change the parameter’s value.

As described above (see Section 6.16.1 [Parameters], page 233), parameters have three flavors; make-shared-parameter, make-thread-parameter, and make-legacy-parameter creates shared, thread, and legacy parameters, respectively.

For the compatibility, make-parameter behaves differently depending on which module you uses.

- The make-parameter in gauche module works as make-legacy-parameter in 0.9.13, but it will be switched to make-shared-parameter in future versions. To keep the legacy code working, you should either use gauche.parameter module, or replace make-parameter for make-legacy-parameter.
- If you use gauche.parameter, make-parameter works as make-legacy-parameter. This won’t change, since gauche.parameter is kept for the backward compatibility.
- If you use srfi.226, make-parameter is as SRFI-226 defines, that is, make-shared-parameter.

NB: R7RS defines make-parameter. SRFI-226 defines make-parameter and make-thread-parameter. The other two names, make-shared-parameter and make-legacy-parameter, are Gauche specific.

NB: In 0.9.9 and before, this procedure returns a parameter object, and we used object-apply method to make it behave like a procedure. However, R7RS explicitly defines the return values to be a procedure; notably, portable code expects (procedure? (make-parameter 'z)) returns #t. As of 0.9.10, we switched make-parameter to return a procedure. It won’t change the external behavior, except when you test the parameter p with (is-a? p <parameter>); you have to use parameter? instead.
parameterize ((param value) ...) body ...

[Macro]

[R7RS base][SRFI-226] Evaluates body ..., with change parameter param's value to the given value within the dynamic scope of body .... Returns the value(s) of the result of the last body.

If parameterize form is at a tail position and all params are parameter objects, body is also evaluated at a tail position. (This is required by SRFI-226, though not required by R7RS).

Note that R7RS and SRFI-226 requires param to be evaluated to a parameter. Gauche’s parameterize has been accepting dynamic states as well, but such usage is deprecated; you should use temporarily instead (see Section 6.16.2 [Dynamic states], page 236). To ease transition, Gauche 0.9.13 checks if every param yields a parameter, at runtime, and if not, it switches to the legacy implementation which does not evaluate body at a tail position. In future we remove this compatibility feature. If you want to check your code is compatible to the new spec, add (use srfi.226) to your code.

Some examples:

```
(define a (make-parameter 1))
(a) ⇒ 1
(a 2) ⇒ 1
(a) ⇒ 2
(parameterize ((a 3))
  (a)) ⇒ 3
(a) ⇒ 2
```

parameter? obj

[SRFI-226] Returns #t iff obj is an object created by make-parameter.

current-parameterization

[Function]

[SRFI-226] Returns a parameterization object that reifies the current dynamic bindings of parameters. It can be passed to call-with-parameterization to carry over the parameterization.

parameterization? obj

[SRFI-226] Returns #t if obj is a parameterization, #f otherwise.

call-with-parameterization parameterization thunk

[Function]

[SRFI-226] Invoke thunk with a continuation same as the call of call-with-parameterization except its parameterization is swapped with parameterization. That is, if the call to this form is in a tail context, the call to thunk is also a tail context.

parameterize/dynwind ((param value) ...) body ...

[Macro]

This handles the parameter bindings using dynamic-wind. It is fully compatible with pre-0.9.13 parameterize. This form is mainly provided to make legacy code work.

This differs from the current parameterize in the following points:

- You can use non-parameter procedure (a procedure which isn’t created by make-parameter) as param, as long as it follows the parameter protocol. Note that if you want to use such 'parameter-like' procedures to manage dynamic states, you should use temporarily below (see Section 6.16.2 [Dynamic states], page 236).
- The last expression of body is not evaluated in the tail context, even this form itself is in the tail context.
- This can handle the 'observer' feature of the legacy parameters introduced by gauche.parameter. Our experience shows the feature has been hardly used, and is incompatible with SRFI-226 semantics, so we deprecated it.
Note: If you use gauche.parameter module, it shadows built-in parameterize with its own parameterize, the latter of which is an alias of parameterize/dynwind, so that the compatibility is kept. New code should not use gauche.parameter module. See Section 9.24 [Parameters (legacy)], page 480.

6.16.2 Dynamic states

A dynamic state is a state associated to a certain dynamic extent of the code execution. (In SRFI-226, it is called “parameter-like objects”).

It can be managed with a procedure that follows this protocol:

- The procedure can be called with zero or one argument.
- If it is called without arguments, it returns the current state.
- If it is called with one argument, the state is set to the argument. (The return value is ignored).

You can use temporarily macro in place of parameterize to change the state dynamically.

```
(define state
  (let val 0
    (case-lambda
      [() val]
      [(newval) (set! val newval)])))

(define (get-state) (state))

(get-state) ⇒ 0

(temporarily ([state 1])
  (get-state)) ⇒ 1
```

If you just manage a value, it is not much different from a parameter (without a converter) and parameterize. However, unlike parameterize, it is guaranteed that state is reset by calling the dynamic state with the previous value, when the control goes out of the dynamic scope. You can do more than just changing the internal values inside the procedure, such as switching state of external resource.

```
temporarily ((proc value) ...) body ...
```

[SRFI-226] Evaluates procs and values. Each proc must yield a dynamic state, i.e. a procedure that takes zero or one arguments.

Then the current state each proc represents is changed to value during evaluating body .... The state is restored when the control leaves body .... The results of the last expression of body will be returned.

Whenever control trasfers out of body, or reenters to body, the state of each proc is saved and restored.

6.16.3 Difference of parameters and dynamic states

At a first glance, parameters and dynamic states look almost same: both are procedures taking zero or one arguments; when called with no arguments, it returns the “current” value; when called with one argument, it replaces the “current” value with the given argument.

The difference is that the “state” of a parameter, or a parameterization, is tied to a continuation. If a continuation frame where parameterize is evaluated is popped, the parameterization simply goes away with it, and the previous parameterization (tied to the previous continuation) automatically becomes visible, instead of the explicit operation of restoring the previous value.
That’s why the body of `parameterize` can be evaluated in tail context—we don’t have a restore operation to perform after the body is evaluated.

On the other hand, dynamic states rely on the ‘after’ handler of `dynamic-wind` to restore the state, hence the body in `temporarily` can’t be evaluated in a tail context.

You want to use dynamic states when the change of state requires some actions other than storing a value, e.g. notifying the change to a resource manager. When a dynamic state procedure is called with a new value, you can take necessary actions. Restoration of parameter values after leaving a dynamic extent is implicit; you can’t know when the parameter value is restored.

Parameters can have converter procedure to coerce the value to be set. Dynamic states aren’t supposed to have the converter procedure, though. If the dynamic extent is inactivated and then resumed by continuations, the dynamic value needs to be restored without passing through the converter procedure, so it requires additional protocol, which isn’t standardized.

### 6.16.4 Transition to SRFI-226 parameters

**First, here’s rule of thumb:**

- If you bind parameters only with `parameterize`, you won’t see any changes.
- If the code explicitly use `gauche.parameter`, it’ll preserve the previous behavior. This helps legacy code to keep working.
- For other code, you can gradually revise your code as follows:
  - If you used parameters as thread-local storage, replace it with thread locals (see Section 9.35.3 [Thread local storage], page 535).
  - If you manage dynamic state with `parameterize`, replace it with `temporarily` (see Section 6.16.2 [Dynamic states], page 236).
  - If you used parameters for both thread-local storage and dynamic binding, use `make-thread-parameter` and `parameterize/dynwind`, in place of `make-parameter` and `parameterize`. (see Section 6.16.1 [Parameters], page 233).

**Under the hood:**

Here we call the parameter implementation before 0.9.13 as “legacy model”, while the one in 0.9.13 and later as “new model”.

In the legacy model, parameterization is realized by swapping each parameter’s value in dynamic wind’s before and after thunk. Roughly speaking, `parameterize` is expanded as follows (this assumes Gauche’s parameter protocol that returns previous value when a new value is set. Also, we omit consideration of converter, for simplicity):

```scheme
(parameterize ((param expr)) body ...)
≡
(let ((p param) (v expr))
    (dynamic-wind
       (lambda () (set! v (p v)))
       (lambda () body ...)
       (lambda () (set! v (p v)))))
```

This model directly replaces the value of the parameter in the dynamic scope of `body` .... So the parameter value needs to be thread-specific, otherwise the swapping of the value would be visible from other threads, which is not desirable.

Note that even parameter value is thread-specific, you can still invoke a continuation captured in a different thread and expect the parameter values are carried over, for the switching
continuation triggers execution of before/after thunks, which restores the values of parameters in the thread that invokes the continuation.

Also keep in mind that the body can’t be in tail context even the parameterize form is, since you have to execute after thunk after the body is evaluated.

In the new model, we keep a chain of dynamic binding frames, and parameter value is looked up from it. The dynamic chain is a part of continuation, so when a continuation is popped, the dynamic frame created in the current continuation frame is also popped implicitly. With this model, parameterize is expanded something like this:

\[
\text{parameterize} \ ((\text{param} \ \text{expr})) \ \text{body} \ldots
\]

\[
\equiv
\]

\[
\lambda (p \ v)
\begin{align*}
& \text{push-dynamic-frame!} \ (p \ v) \\
& \text{body} \ldots \\
& \text{param} \ \text{expr}
\end{align*}
\]

Here, push-dynamic-frame! is a hypothetical procedure that pushes new dynamic frame in the current continuation. When the control returns to the continuation of the parameterize form, the additional dynamic frame is popped automatically so we don’t need a special operation. If a continuation is captured in body, the dynamic frame chain is saved along the continuation. When the continuation is invoked in another thread, the dynamic frame chain is restored. This allows the body to be run in tail context if parameterize itself is in tail context.

If you mutate parameter value (as opposed to binding it with parameterize), the dynamic frame is mutated. If the frame is shared between multiple threads, those mutations are visible immediately; they’re pretty much the same as lexical frames, except the scopes.

SRFI-226 thread parameters are required to contain mutations within a thread. So we create a new thread-local whenever a thread parameter is dynamically bound by parameterize and keep the parameter’s value in it. The mutation on a thread parameter is never visible from other thread, even if the dynamic frames are shared, or a continuation having it is passed to another thread. So, be careful using thread parameters. For example, if you create a future (see Section 12.7 [Futures], page 816), the expression in it is evaluated in another thread, so it may not see the same value in thread parameters as the caller. Thread parameters also incur overhead for every parameterize, to allocate a new thread local.

The new model also restricts parameterize to bind only parameters. “Parameter-like” objects, a procedure that merely follows the parameter protocol, can’t be handled in parameterize. You should separate them and use temporarily for the latter (see Section 6.16.2 [Dynamic states], page 236).

To ease transition, 0.9.13’s parameterize checks if the given “parameters” are all actual parameters at run-time, and if not, it evaluates as if the form is parameterize/dynwind instead. This kind of loose behavior tends to be the source of bugs, so we’ll eventually make it stricter. You can check if your code is conforming the new model by using (use srfi.226.parameter).

### 6.17 Boxes

A box is a mutable container that can hold (possibly multiple) values. It can be used as a minimal data storage, or a sort of mutable indirect “pointer”.

Traditionally, a list or a vector has been used for this purpose. However, such datatypes imply sequences of objects, Using boxes emphasizes your intention is just for indirection, and not so much for sequencing.

It is originally introduced by SRFI-111, which was later adopted in R7RS-large as scheme.box. SRFI-195 enhances it to deal with multiple values.
The srfis leave some details to implementations. Here are our choices:

- We don’t support autoboxing; that is, it is an error to pass non-box value to the procedure expecting boxed value and vice versa.
- Comparing two boxes with equal? compares their contents when two are not eqv?. In the spec, when two boxes are eqv? then they must also be equal? to each other, but it’s up to the implementation when two are not eqv?.

When you’re writing portable code, be careful not to depend on the equal? behavior.

```scheme
box val ...
[Function] [R7RS box][SRFI-195] Returns a fresh box object that contains the value val ...

box? obj
[Function] [R7RS box] Returns #t iff obj is a box object.

box-arity box
[Function] [SRFI-195] Returns the number of values box holds.

unbox box
[Function] [R7RS box] Returns box’s content. If box has N values, it returns N values.

unbox-value box i
[Function] [SRFI-195] Returns i-th value held in box.

set-box! box val ...
[Function] [R7RS box][SRFI-195] Alters the content of box with val ... The number of values must match the arity of the box. Returns unspecified value.

set-box-value! box i val
[Function] [SRFI-195] Alters i-th value of box with val. Returns unspecified value.
```

6.18 Lazy evaluation

Gauche has two primitive lazy evaluation mechanisms.

The first one is an explicit mechanism, defined in the Scheme standard: You mark an expression to be evaluated lazily by delay, and you use force to make the evaluation happen when needed. Gauche also support another primitive lazy, as defined in SRFI-45, for space-efficient tail-recursive lazy algorithms.

The second one is a lazy sequence, in which evaluation happens implicitly. From a Scheme program, a lazy sequence just looks as a list—you can take its car and cdr, and you can apply map or other list procedures on it. However, internally, its element isn’t calculated until it is required.

6.18.1 Delay, force and lazy

Scheme has traditionally provided an explicit delayed evaluation mechanism using delay and force. After R5RS, however, it is found that it didn’t mix well with tail-recursive algorithms: It required unbound memory, despite that the body of the algorithm could be expressed in iterative manner. SRFI-45 showed that introducing another primitive syntax lazy addresses the issue. For the detailed explanation please look at the SRFI-45 document. Here we explain how to use those primitives.

```scheme
delay expression
[Special Form]
lazy expression
[Special Form] [R7RS lazy][SRFI-45] These forms creates a promise that delays the evaluation of expression. Expression will be evaluated when the promise is passed to force.
```
If `expression` itself is expected to yield a promise, you should use `lazy`. Otherwise, you should use `delay`. If you can think in types, the difference may be clearer.

```scheme
lazy : Promise a -> Promise a
delay : a -> Promise a
```

Since we don’t have static typing, we can’t enforce this usage. The programmer has to choose appropriate one from the context. Generally, `lazy` appears only to surround the entire body of function that express a lazy algorithm.

NB: In R7RS, `lazy` is called `delay-force`, for the operation is conceptually similar to `(delay (force expr))` (note that the type of `force` is `Promise a -> a`).

For the real-world example of use of `lazy`, you may want to check the implementation of `util.stream` (see Section 12.90 [Stream library], page 1033).

Note that `expression` may be evaluated more than once; it may require the value of its own promise recursively, or more than one thread may evaluate `expression` simultaneously. In general, you shouldn’t have side-effecting computation in `expression`—even if it is only evaluated once, you may not exactly know when the evaluation occur, so having side-effects there is a bad ides. (See `force` below for multi-thread semantics).

```scheme
[Function] 
eager obj ...

[Function] 
make-promise obj ...

[SRFI-45][SRFI-226] Returns a promise that returns `obj`. The name `eager` is introduced in SRFI-45. It takes single argument, but Gauche enhanced it to take multiple values. SRFI-226 defines `make-promise`, which is the same as multi-value `eager`. (Note that R7RS-small also defines `make-promise` in `scheme.lazy` library (see Section 10.2.10 [R7RS lazy evaluation], page 590). It is slightly different that it only takes one argument, and if the argument is a promise, it will be returned as is. This change is intentional; see SRFI-226 for the details.)

Since that these are procedures, arguments are evaluated before they are called; so they work as a type converter (`a -> Promise a`) without delaying the evaluation. Used mainly to construct promise-returning functions.

```scheme
[Function] 
force promise

[SRFI-226] If `promise` is not a promise, it is just returned.

Otherwise, if `promise`’s value hasn’t been computed, `force` makes `promise`’s encapsulated expression be evaluated in the same parameterization where the promise is created, and returns the result.

Once `promise`’s value is computed, it is memorized in it so that subsequent `force` on it won’t cause the computation.

In Gauche, if the delayed expression yields multiple values, `force` yields multiple values as well. (It is unspecified in R7RS).

Note that if uncomputed promise is forced simultaneously by more than one thread, its expression may be evaluated concurrently. The first delivered value becomes the value of the promise; if another thread finishes computation and see the value of promise is already determined, it discards its computation result and returns the determined value.

The following example shows the promise is evaluated with the parameterization of `delay`:

```scheme
(define p (make-parameter 1))

(let1 v (parameterize ((p 2)) (delay (p)))
 (parameterize ((p 3))
  (force v)))
⇒ 2
```
Note: R7RS defines \texttt{force} to evaluate the unevaluated promise's body in the dynamic environment of the caller of \texttt{force}, but it is regarded as a mistake. In general, you shouldn't care which \texttt{force} call actually evaluates the promise's body, so the result shouldn't depend on the context of \texttt{force} call. SRFI-226 addressed it.

\begin{function}
\texttt{promise? obj} \hfill \text{(Function)}
\end{function}

\text{[R7RS lazy]} Returns \texttt{#t} iff \texttt{obj} is a promise object.

\section*{6.18.2 Lazy sequences}

\subsection*{Introduction}

A lazy sequence is a list-like structure whose elements are calculated lazily. Internally we have a special type of pairs, whose \texttt{cdr} is evaluated on demand. However, in Scheme level, you'll never see a distinct "lazy-pair" type. As soon as you try to access a lazy pair, Gauche automatically \texttt{force} the delayed calculation, and the lazy pair turns into an ordinary pair.

It means you can pass lazy sequences to ordinary list-processing procedures such as \texttt{car}, \texttt{cdr} or \texttt{map}.

Look at the following example; \texttt{generator->lseq} takes a procedure that generates one value at a time, and returns a lazy sequence that consists of those values.

\begin{verbatim}
(with-input-from-file "file"
 (\[] (let loop ([cs (generator->lseq read-char)] [i 0])
     (match cs
         [() #f]
         [(#\c (or #\a #\d) #\r . _) i]
         [(c . cs) (loop cs (+ i 1))])))
)
\end{verbatim}

It returns the position of the first occurrence of character sequence "car" or "cdr" in the file \texttt{file}. The loop treats the lazy sequence just like an ordinary list, but characters are read as needed, so once the sequence is found, the rest of the file won't be read. If we do it eagerly, we would have to read the entire file first no matter how big it is, or to give up using the mighty \texttt{match} macro and to write a basic state machine that reads one character one at a time.

Other than implicit forcing, Gauche's lazy sequences are slightly different than the typical lazy stream implementations in Scheme in the following ways:

1. When you construct a lazy sequence in an iterative lazy algorithm, only \texttt{cdr} side of the lazy pair is lazily evaluated; the \texttt{car} side is evaluated immediately. On the other hand, with \texttt{stream-cons} in \texttt{util.stream} (see Section 12.90 [Stream library], page 1033), both \texttt{car} and \texttt{cdr} sides won't be evaluated until it is absolutely needed.

2. Gauche's lazy sequence always evaluates \texttt{one item ahead}. Once you get a lazy pair, its \texttt{car} part is already calculated, even if you don't use it. In most cases you don't need to care, for calculating one item more is a negligible overhead. However, when you create a self-referential lazy structure, in which the earlier elements of a sequence is used to calculate the latter elements of itself, a bit of caution is needed; a valid code for fully lazy circular structure may not terminate in Gauche's lazy sequences. We'll show a concrete example later. This bit of eagerness is also visible when side effects are involved; for example, lazy character sequence reading from a port may read one character ahead.

Note: R7RS \texttt{scheme.lseq} (SRFI-127) provides a portable alternative of lazy sequence (see Section 10.3.13 [R7RS lazy sequences], page 633). It uses dedicated APIs (e.g. \texttt{lseq-cdr}) to operate on lazy sequences so that portable implementation is possible. In Gauche, we just use our built-in lazy sequence as SRFI-127 lazy sequence; if you want your code to be portable, consider using SRFI-127, but be careful not to mix lazy sequences and ordinary lists; Gauche won't complain, but other Scheme implementation may choke on it.
Primitives

generator->lseq generator

[Function]

[R7RS lseq] Creates a lazy sequence that consists of items produced by generator, which is just a procedure with zero arguments that yields an item at a time. Returning EOF marks the end of the sequence (EOF itself isn’t included in the sequence). For example, read-char can work as a generator. Gauche has a set of convenient utilities to deal with generators (see Section 9.12 [Generators], page 436).

In the second form, the returned lazy sequence is prepended by item .... Since there’s no way to distinguish lazy pairs and ordinary pairs, you can write it as (cons* item ... (generator->lseq generator)), but that’s more verbose.

Internally, Gauche’s lazy sequence is optimized to be built on top of generators, so this procedure is the most efficient way to build lazy sequences.

Note: SRFI-127 also has generator->lseq, which is exactly the same as this in Gauche.

lcons car cdr

Returns a lazy pair consists of car and cdr. The expression car is evaluated at the call of lcons, but evaluation of cdr is delayed.

You can’t distinguish a lazy pair from an ordinary pair. If you access either its car or cdr, or even you ask pair? to it, its cdr part is implicitly forced and you get an ordinary pair.

Unlike cons, cdr should be an expression that yields a (lazy or ordinary) list, including an empty list. In other words, lazy sequences can always be a null-terminated list when entirely forced; there are no “improper lazy sequences”. (Since Scheme isn’t statically typed, we can’t force the cdr expression to be a proper list before actually evaluating it. Currently if cdr expression yields non-list, we just ignore it and treat as if it yielded an empty list.)

(define z (lcons (begin (print 1) 'a) (begin (print 2) '())))
⇒ ; prints '1', since the car part is evaluated eagerly.

(cdr z) ⇒ () ;; and prints '2'

;; This also prints '2', for accessing car of a lazy pair forces
;; its cdr, even the cdr part isn’t used.
(car (lcons 'a (begin (print 2) '()))) ⇒ a

;; So as this; asking pair? to a lazy pair causes forcing its cdr.
(pair? (lcons 'a (begin (print 2) '()))) ⇒ #t

;; To clarify: This doesn’t print '2', because the second lazy
;; pair never be accessed, so its cdr isn’t evaluated.
(pair? (lcons 'a (lcons 'b (begin (print 2) '(1)))))) ⇒ #t

Now, let me show you a case where “one item ahead” evaluation becomes an issue. The following is an elegant definition of infinite Fibonacci sequence using self-referential lazy structure (lmap is a lazy map, defined in gauche.lazy module):

(use gauche.lazy) ;; for lmap
(define *fibs* (lcons 0 1 (lmap + *fibs* (cdr *fibs*)))) ;; BUGGY

Unfortunately, Gauche can’t handle it well.

(car *fibs*)
⇒ 0

(cadr *fibs*)
⇒ *** ERROR: Attempt to recursively force a lazy pair.

When we want to access the second argument (cadr) of *fibs*, we take the car of the second pair, which is a lazy pair of 1 and (lmap ...). The lazy pair is forced and its cdr part needs to be calculated. The first thing lmap returns needs to see the first and second element of *fibs*, but the second element of *fibs* is what we’re calculating now!

We can workaround this issue by avoiding accessing the immediately preceding value. Fibonacci numbers $F(n) = F(n-1) + F(n-2) = 2F(n-2) + F(n-3)$, so we can write our sequence as follows.

```
(define *fibs*
  (lcons* 0 1 1 (lmap (^[a b] (+ a (* b 2))) *fibs* (cdr *fibs*))))
```

And this works!

```
(take *fibs* 20)
⇒ (0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181)
```

Many lazy algorithms are defined in terms of fully-lazy cons at the bottom. When you port such algorithms to Gauche using lcons, keep this bit of eagerness in mind.

Note also that lcons needs to create a thunk to delay the evaluation. So the algorithm to construct lazy list using lcons has an overhead of making closure for each item. For performance-critical part, you want to use generator->lseq whenever possible.

**Utilities**

`lcons* x ... tail`  
[Macro]

`llist* x ... tail`  
[Macro]

A lazy version of cons* (see Section 6.6.4 [List constructors], page 142). Both lcons* and llist* do the same thing; both names are provided for the symmetry to cons*/list*.

The tail argument should be an expression that yields a (possibly lazy) list. It is evaluated lazily. Note that the preceding elements x ... are evaluated eagerly. The following equivalences hold.

```
(lcons* a)       ≡ a
(lcons* a b)     ≡ (lcons a b)
(lcons* a b ... y z) ≡ (cons* a b ... (lcons y z))
```

**lrange start :optional end step**  
[Function]

Creates a lazy sequence of numbers starting from start, increasing by step (default 1), to the maximum value that doesn’t exceed end. The default of end is +inf.0, so it creates an infinite list. (Don’t type just (lrange 0) in REPL, or it won’t terminate!)

If any of start or step is inexact, the resulting sequence has inexact numbers.

```
(take (lrange -1) 3) ⇒ (-1 0 1)
```

```
(lrange 0.0 5 0.5)
⇒ (0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5)
```

```
(lrange 1/4 1 1/8)
⇒ (1/4 3/8 1/2 5/8 3/4 7/8)
```

**liota :optional (count +inf.0) (start 0) (step 1)**  
[Function]

A lazy version of iota (see Section 6.6.4 [List constructors], page 142); returns a lazy sequence of count integers (default: positive infinity), starting from start (default: 0), stepping by step (default: 1).
Just like \texttt{iota}, the result consists of exact numbers if and only if both \texttt{start} and \texttt{step} are exact; otherwise the result consists of inexact numbers.

\begin{verbatim}
port->char-lseq :optional port [Function]
port->byte-lseq :optional port [Function]
port->string-lseq :optional port [Function]
port->sexp-lseq :optional port [Function]
\end{verbatim}

These are the same as the following expressions, respectively. They are provided for the convenience, since this pattern appears frequently.

\begin{verbatim}
(generator->lseq (cut read-char port))
(generator->lseq (cut read-byte port))
(generator->lseq (cut read-line port))
(generator->lseq (cut read port))
\end{verbatim}

If \texttt{port} is omitted, the current input port is used.

Note that the lazy sequence may buffer some items, so once you make an lseq from a port, only use the resulting lseq and don’t ever read from \texttt{port} directly.

Note that the lazy sequence terminates when EOF is read from the port, but the port isn’t closed. The port should be managed in larger dynamic extent where the lazy sequence is used.

You can also convert input data into various lists by the following expressions (see Section 6.21.7.4 [Input utility functions], page 274). Those procedures read the port eagerly until EOF and returns the whole data in a list, while \texttt{lseq} versions read the port lazily.

\begin{verbatim}
(port->list read-char port)
(port->list read-byte port)
(port->string-list port)
(port->sexp-list port)
\end{verbatim}

Those procedures make (lazy) lists out of ports. The opposite can be done by \texttt{open-input-char-list} and \texttt{open-input-byte-list}; see Section 9.40 [Virtual ports], page 572, for the details.

See also Section 9.15 [Lazy sequence utilities], page 451, for more utility procedures that creates lazy sequences.

\section*{Examples}

Let’s consider calculating an infinite sequence of prime numbers. (Note: If you need prime numbers in your application, you don’t need to write one; just use \texttt{math.prime}. see Section 12.37 [Prime numbers], page 895).

Just pretend we already have some prime numbers calculated in a variable \texttt{*primes*}, and you need to find a prime number equal to or grater than \texttt{n} (for simplicity, we assume \texttt{n} is an odd number).

\begin{verbatim}
(define (next-prime n)
  (let loop ([ps *primes*])
    (let1 p (car ps)
      (cond [(> (* p p) n) n]
            [(zero? (modulo n p)) (next-prime (+ n 2))]
            [else (loop (cdr ps))])))
\end{verbatim}

This procedure loops over the list of prime numbers, and if no prime number \texttt{p} less than or equal to \texttt{(sqrt n)} divides \texttt{n}, we can say \texttt{n} is prime. (Actual test is done by \texttt{(> (* p p) n)} instead of \texttt{(p (sqrt n))}, for the former is faster.) If we find some \texttt{p} divides \texttt{n}, we try a new value \texttt{(+ n 2)} with \texttt{next-prime}.
Using `next-prime`, we can make a generator that keeps generating prime numbers. The following procedure returns a generator that returns primes above `last`.

```
(define (gen-primes-above last)
  ([] (set! last (next-prime (+ last 2))) last))
```

Using `generator->lseq`, we can turn the generator returned by `gen-primes-above` into a lazy list, which can be used as the value of `*prime*`. The only caveat is that we need to have some pre-calculated prime numbers:

```
(define *primes* (generator->lseq 2 3 5 (gen-primes-above 5)))
```

Be careful not to evaluate `*primes*` directly on REPL, since it contains an infinite list and it'll blow up your REPL. You can look at the first 20 prime numbers instead:

```
(take *primes* 20)
⇒ (2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71)
```

Or find what the 10000-th prime number is:

```
(take *primes* 10000)
⇒ 104743
```

Or count how many prime numbers there are below 1000000:

```
(any (^[p i] (and (>= p 1000000) i)) *primes* (lrange 0))
⇒ 78498
```

Note: If you’re familiar with the lazy functional approach, this example may look strange. Why do we use side-effecting generators while we can define a sequence of prime numbers in pure functional way, as follows?

```
(use gauche.lazy)

(define (prime? n)
  (not (any (^p (zero? (mod n p)))
            (ltake-while (^k (<= (* k k) n)) *primes*))))

(define (primes-from k)
  (if (prime? k)
      (lcons k (primes-from (+ k 2)))
      (primes-from (+ k 2))))

(define *primes* (llist* 2 3 5 (primes-from 7)))
```

(The module `gauche.lazy` provides `ltake-while`, which is a lazy version of `take-while`. We don’t need lazy version of `any`, since it immediately stops when the predicate returns a true value.)

The use of `lcons` and co-recursion in `primes-from` is a typical idiom in functional programming. It’s perfectly ok to do so in Gauche; except that the generator version is much faster (when you take first 5000 primes, generator version ran 17 times faster than co-recursion version on the author’s machine).

It doesn’t mean you should avoid co-recursive code; if an algorithm can be expressed nicely in co-recursion, it’s perfectly ok. However, watch out the subtle semantic difference from lazy functional languages—straightforward porting may or may not work.

### 6.19 Exceptions

Gauche’s exception system consists of three components; (1) the way to signal an exceptional case has occurred, (2) the way to specify how to handle such a case, and (3) the standard objects
(conditions) to communicate the code that signals an exceptional case and the code that handles it.

Those three components are typically used together, so first we explain the typical usage patterns using examples. Then we describe each feature in detail.

Note for terminology: some languages use the word exception to refer to an object used to communicate the code that encountered an exceptional situation with a handler that deals with it. Gauche uses a term condition to refer to such objects, following SRFI-35. Exception is the situation, and condition is a runtime object that describes it.

6.19.1 Exception handling overview

Catching specific errors

One of the most typical exception handling is to catch a specific error raised by some built-in or library procedures. A macro guard can be used for such a purpose. The code looks like this:

```lisp
(guard (exc [(condition-has-type? exc <read-error>)
             (format #t "read error!")
             'read-error]
             [else 'other-error])
      (read-from-string "(abc")
```

The cadr of guard clause is a form of (variable clause ...). In this example, the variable is exc, and it has two clauses. Each clause has the form like the one in cond.

The cddr of guard is the body, a list of expressions. This example has only one expression, (read-from-string "(abc")).

**guard** starts executing its body. read-from-string raises an error of type <read-error> when it encounters syntactic errors. The form guard intercepts the error, and binds the condition object to the variable exc, then checks the clauses following exc in a similar manner to cond—in this case, the thrown condition is of type <read-error>, so the test of the first clause is satisfied, and the rest of clause is executed, i.e. "read error!" is printed and a symbol read-error is returned.

If you’re familiar with other languages, you may recognize the pattern. The cddr of guard form is like try clause of C++/Java or the cadr of handler-case of Common Lisp; and the cdadr of guard form is like catch clauses or the cddr of handler-case.

In the test expressions it is common to check the type of thrown condition. The function condition-has-type? is defined in SRFI-35 but it’s rather lengthy. Gauche’s condition classes can also work like a predicate, so you can write the above expression like this.

```lisp
(guard (exc [(<read-error> exc)
              (format #t "read error!")
              'read-error]
              [else 'other-error])
       (read-from-string "(abc")
```

Note: Generally you can’t use is-a? to test if the thrown condition is of a specific type, since a condition may be compound. See Section 6.19.4 [Conditions], page 253, about compound conditions.

If no tests of clauses satisfy and no else clause is given, the exception ‘falls off’ the guard construct, i.e. it will be handled by the outer level of guard form or top-level. For example, the following guard form only handles <read-error> and <system-error>; if the body throws other type of conditions, it must be handled by outer level.

```lisp
(guard (exc [[<read-error> exc] (handle-read-error)]
            [(<system-error> exc) (handle-system-error)])
```

body ...)

See Section 6.19.3 [Handling exceptions], page 249, for more details on guard and other lower-level exception handling constructs.

**Signaling exceptions from your code**

The generic way to signal an exception is to use raise procedure.

```scheme
(raise condition)
```

You can pass any object to condition; its interpretation solely depends on the exception handler. If you know the code raises an integer as a condition, you can catch it by guard as this:

```scheme
(guard (exc [(integer? exc) 'raised]))
(raise 3))
```

However, as a convention, it is preferable to use an instance of condition or one of its subclasses. A macro condition can be used to create a condition object. The following examples show how to create a condition with some slot values and then raise it.

```scheme
;;; create and raise an error condition
(raise (condition
  (<error> (message "An error occurred.")))
)

;;; create and raise a system error condition
(raise (condition
  (<system-error> (message "A system error occurred.")
    (errno EINTR)))
)
```

See Section 6.19.4 [Conditions], page 253, for the details of condition macro and what kind of condition classes are provided.

The most common type of condition is an error condition, so a convenience procedure error and errorf are provided. They create an error condition with a message and raise it.

```scheme
;;; 'error' concatenates the arguments into a message.
(unless (integer? obj)
  (error "Integer expected, but got:" obj))

;;; 'errorf' uses format to create a message.
(unless (equal? x y)
  (errorf "%s and %s don't match" x y))
```

Unlike the exception throwing constructs in some languages, such as throw of C++/Java, which abandons its continuation, Scheme’s raise may return to its caller. If you don’t want raise to return, a rule of thumb is always to pass one of error conditions to it; then Gauche guarantees raise won’t return. See the description of raise in Section 6.19.2 [Signaling exceptions], page 248, for more details.

Note: R7RS adopted slightly different semantics; it splits raise and raise-continuable, the former is for noncontinuable exception (if the exception handler returns, it raises another error), and the latter is for continuable exception. When you’re in R7RS environment, R7RS-compatible raise will be used instead of this raise.

**Defining your own condition**

You can also define your own condition classes to pass application-specific information from the point of raising exception to the handlers.

To fit to Gauche’s framework (SRFI-35), it is desirable that the new condition class inherits a built-in condition class or one of its descendants, and also is an instance of a metaclass condition-meta.
One way of ensuring the above convention as well as increasing portability is to use `define-condition-type` macro, defined in `srfi.35`.

```
(define-condition-type <myapp-error> <error>
  myapp-error?
  (debug-info myapp-error-debug-info)
  (reason myapp-error-reason))
```

This defines a condition type (which is a class in Gauche) `<myapp-error>`, with a predicate `myapp-error?` and slots with accessors. Then you can use the new condition type like the following code:

```
(guard (exc
  [(myapp-error? exc)
   (let ([debug-info (myapp-error-debug-info exc)]
      [reason (myapp-error-reason exc)])
     ... handle myapp-error ...)])
  ...
  ...
  (if (something-went-wrong)
    (raise (condition
      (<myapp-error> (debug-info "during processing xxx")
      (reason "something went wrong"))))))
```

If you don’t mind to lose SRFI compatibility, you can use Gauche’s extended `error` and `errorf` procedures to write more concise code to raise a condition of subtype of `<error>`:

```
(if (something-went-wrong)
  (error <myapp-error>
    :debug-info "during processing xxx"
    :reason "something went wrong")
```

See the description of `define-condition-type` macro for how the condition type is implemented in Gauche’s object system.

### 6.19.2 Signaling exceptions

#### Signaling errors

The most common case of exceptions is an error. Two convenience functions to signal an error condition in simple cases are provided. To signal a compound condition, you can use `raise` as explained below.

```
error string arg . . .  [Function]
error condition-type keyword-arg ... string arg . . .  [Function]
[R7RS+][SRFI-23+] Signals an error. The first form creates an `<error>` condition, with a message consists of `string` and `arg` . . . , and raises it. It is compatible to R7RS and SRFI-23’s `error` behavior.

gosh> (define (check-integer x)
  (unless (integer? x)
    (error "Integer required, but got:" x)))
check-integer
gosh> (check-integer "a")
*** ERROR: Integer required, but got: "a"
Stack Trace:
The second form can be used to raise an error other than the `<error>` condition. `condition-type` must be a condition type (see Section 6.19.4 [Conditions], page 253, for more explanation of condition types). It may be followed by keyword-value list to initialize the condition slots, and then optionally followed by a string and other objects that becomes an error message.

```
(define-condition-type <my-error> <error> #f
  (reason)
  (priority))
```

```
... (unless (memq operation *supported-operations*)
  (error <my-error>
    :reason 'not-supported :priority 'urgent
    "Operation not supported:" operation))
...  
```

**Signaling generic conditions**

**raise condition**  
[SRFL-18][R7RS base] This is the base mechanism of signaling exceptions. The procedure invokes the current exception handler. The argument `condition` represents the nature of the exception, and passed to the exception handler. Gauche’s built-in and library functions always use an instance of `<condition>` or one of its subclasses as `condition`, but you can pass any Scheme object to `raise`. The interpretation of `condition` is up to the exception handler.

*Note:* Unlike some of the mainstream languages in which "throwing" an exception never returns, you can set up an exception handler in the way that `raise` may return. The details are explained in Section 6.19.3 [Handling exceptions], page 249.

If you don’t want `raise` to return, the best way is to pass a condition which is an instance of `<serious-condition>` or one of its subclasses. Gauche’s internal mechanism guarantees raising such an exception won’t return. See Section 6.19.4 [Conditions], page 253, for the hierarchy of built-in conditions.

R7RS adopted slightly different semantics regarding returning from `raise`; in R7RS, `raise` never returns—if the exception handler returns, another exception is raised. R7RS has `raise-continuable` to explicitly allow returning from the exception handler. For portable programs, always pass `<serious-condition>` or its subclasses to `raise`.

### 6.19.3 Handling exceptions

#### 6.19.3.1 High-level exception handling mechanism

**guard (var clause ...) body ...**  
[R7RS base] This is the high-level form to handle errors in Gauche.
var is a symbol, and clauses are the same form as cond’s clauses, i.e. each clause can be either one of the following forms:

1. \( \text{test expr ...} \)
2. \( \text{test \ => \ proc} \)

The last clause may be \( \text{else expr ...} \).

This form evaluates body ... and returns the value(s) of the last body expression in normal case. If an exception is raised during the evaluation of body expressions, the raised exception is bound to a variable var, then evaluates test expression of each clause. If one of test expressions returns true value, then the corresponding exprs are evaluated if the clause is the first form above, or an proc is evaluated and the result of test is passed to the procedure proc if the clause is the second form.

When the test(s) and expr(s) in the clauses are evaluated, the exception handler that is in effect of the caller of guard are installed; that is, if an exception is raised again within clauses, it is handled by the outer exception handler or guard form.

If no test returns true value and the last clause is else clause, then the associated exprs are evaluated. If no test returns true value and there’s no else clause, the raised exception is re-raised, to be handled by the outer exception handler.

When the exception is handled by one of clauses, guard returns the value(s) of the last expr in the handling clause.

The clauses are evaluated in the same dynamic environment as the guard form, i.e. any dynamic-winds inside body are unwound before evaluation of the clauses. It is different from the lower level forms with-error-handler and with-exception-handler, whose handler is evaluated before the dynamic environment are unwound.

\[
\text{(let ([z '()])} \\
\text{(guard (e [else (push! z 'caught)])} \\
\text{(dynamic-wind (lambda () (push! z 'pre)))} \\
\text{(lambda () (error "foo")))} \\
\text{(lambda () (push! z 'post)))} \\
\text{(reverse z))} \\
\Rightarrow \text{(pre post caught)}
\]

\[
\text{(guard (e [else (print 'OUTER) #f])} \\
\text{(with-output-to-string} \\
\text{(lambda ()} \\
\text{(print 'INNER) \\
\text{(error "foo")))})} \\
\Rightarrow \text{prints OUTER to the current output port of guard, not to the string port.}
\]

unwind-protect expr cleanup ... [Macro]

Executes expr, then executes cleanups, and returns the result(s) of expr. If an uncontinuable exception is raised within expr, cleanups are executed before the exception escapes from the unwind-protect form. For example, the following code calls start-motor, drill-a-hole, and stop-motor in order if everything goes ok, and if anything goes wrong in start-motor or drill-a-hole, stop-motor is still called before the exception escapes unwind-protect.

\[
\text{(unwind-protect} \\
\text{(begin (start-motor) \\
\text{(drill-a-hole))} \\
\text{(stop-motor))}
\]
The cleanup forms are evaluated in the same dynamic environment as unwind-protect. If an exception is thrown within cleanup, it will be handled outside of the unwind-protect form.

Although this form looks similar to dynamic-wind, they work at different layers and should not be confused. dynamic-wind is the bottom-level building block and used to manage current exception handlers, current i/o ports, parameters, etc. dynamic-wind’s before and after thunks are called whenever any of those control flow transition occurs. On the other hand, unwind-protect only cares about the Gauche’s exception system. unwind-protect’s cleanup is called only when expr exits normally or throws Gauche’s exception. In the above example, if control escapes from drill-a-hole by calling a continuation captured outside of unwind-protect, cleanup is not called; because the control may return to drill-a-hole again. It can happen if user-level thread system is implemented by call/cc, for example.

You can go back to the body expr from outside of unwind-protect by invoking continuations captured within expr.

However, keep in mind that once cleanup are executed, some resources might not be available in expr. We still allow it since the reexecuted part of expr may not depend on the resources cleaned up with cleanup.

Even if expr returns (normally or abnormally), cleanup only executed once, in the first time. The name of this form is taken from Common Lisp. Some Scheme systems have similar macros in different names, such as try-finally.

**with-error-handler** handler thunk  
[Function]  
Makes handler the active error handler and executes thunk. If thunk returns normally, the result(s) will be returned. If an error is signaled during execution of thunk, handler is called with one argument, an exception object representing the error, with the continuation of with-error-handler. That is, with-error-handler returns whatever value(s) handler returns.

If handler signals an error, it will be handled by the handler installed when with-error-handler called.

The dynamic environment where handler is executed is the same as the error occurs. If dynamic-wind is used in thunk, its after method is called after handler has returned, and before with-error-handler returns.

Note: Using this procedure directly is no longer recommended, since guard is more safe and portable. We’ll keep this for a while for the backward compatibility, but we recommend to rewrite code to use guard instead of this. The common idiom of "cleanup on error" code:

```lisp
(with-error-handler (lambda (e) (cleanup) (raise e))
(lambda () body ...))
```

should be written like this:

```lisp
(guard (e [else (cleanup) (raise e)])
  body ...)
```

### 6.19.3.2 Behavior of unhandled exception

If an exception is raised where no program-defined exception handler is installed, the following action is taken.

1. If an unhandled exception occurs within a thread other than the primordial one, it terminates the thread, and the thrown condition is wrapped by `<uncaught-exception>` condition and stored in the thread object. If other thread calls thread-join! to retrieve result, the the `<uncaught-exception>` is thrown in that thread. Note that no messages are displayed when the original uncaught exception is thrown. See Section 9.35.1 [Thread programming tips], page 530, for the details.
2. Otherwise, if the program is running interactively (in repl), the information of the thrown exception and stack trace are displayed, and the program returns to the toplevel prompt.

3. If the program is running non-interactively, the information of the thrown exception and stack trace are displayed, then the program exits with an exit status EX_SOFTWARE (70).

The default error message and stack trace in the above case 2 and case 3 is printed by `report-error` procedure. You can use it in your error handler if you need the same information.

```plaintext
report-error exn :optional sink
```

Prints type and message of a thrown condition object `exn`, then print the current stack trace. This is the procedure the system calls when you see an error reported on REPL.

Since you can `raise` any object, `exn` can be any object; it’s not needed to be an instance of `<condition>`. A suitable message is chosen by `report-error`.

You can specify where the output goes by the optional `sink` argument: If it is an output port, the output goes there; you can also pass `#t` for the current output port and `#f` for the output string port, just like `format`. That is, when you pass `#f`, the message goes to a temporary output string port, and gathered string is returned. For all the other cases, an undefined value is returned. If `sink` is omitted or any other object listed above, the current error port is used.

Note: As of 0.9.5, this procedure prints stack trace of the context where `report-error` is called, rather than the context where `exn` is thrown. It doesn’t matter much as far as you call `report-error` directly inside the error handler, but in general what you want to print is the latter, and we have a plan to attach stack trace info to `<condition>` object in future.

### 6.19.3.3 Low-level exception handling mechanism

This layer provides SRFI-18 compatible simple exception mechanism. You can override the behavior of higher-level constructs such as `with-error-handler` by using `with-exception-handler`.

Note that it is a double-edged sword. You’ll get a freedom to construct your own exception handling semantics, but the Gauche system won’t save if something goes wrong. Use these primitives when you want to customize the system’s higher-level semantics or you are porting from other SRFI-18 code.

```plaintext
current-exception-handler
```

[SRFI-18] Returns the current exception handler.

```plaintext
exception-handler-stack
```

[SRFI-226] Returns a fresh list of handlers in the current exception handler stack. The most recent one coming first.

```plaintext
with-exception-handler handler thunk
```

[R7RS base][SRFI-34] A procedure `handler` must take one argument. This procedure sets `handler` to the current exception handler and calls `thunk`. (Note that this slightly differs from SRFI-18 `with-exception-handler`; we’ll explain it below.)

When an exception is raised by `raise` or `error`, `handler` is called with the thrown condition in the exactly same dynamic environment of `raise` or `error`, except that the exception handler at the time `with-exception-handler` is called is restored.

Note: SRFI-18 specifies the exception handler installed with `with-exception-handler` will be called with exactly the same dynamic environment, including the exception handler settings. It means if an exception is raised within the handler, it will be caught with the same handler. The reasoning is that `with-exception-handler` is the bottom layer and kept as simple as possible, and further semantics should be built on top of it.
Until 0.9.10 we supported SRFI-18 semantics natively, and provided R7RS semantics on top of it. However, the bare SRFI-18 semantics turned out to be error prone–users tended to assume errors from a handler would be handled by outer handler, and were perplexed when they ran into infinite recursion of handlers (which eventually caused a segfault, for the recursion eats up C stack). We decided to switch to R7RS semantics, for it is virtually always what users want.

If an exception is raised by error, or the thrown condition inherits <serious-condition>, it is prohibited to return from handler. If handler ever returns in such cases, another error is signaled, with replacing the current exception handler to the outer handler. So the caller of error, or the caller of raise with <serious-condition>, can assume it never returns.

The behavior of those procedures can be explained in the following conceptual Scheme code.

```scheme
;; Conceptual implementation of low-level exception mechanism.
;; Suppose %xh is a list of exception handlers

(define (current-exception-handler) (car %xh))

(define (raise exn)
  (let ((prev %xh))
    (dynamic-wind
     (lambda () (set! %xh (cdr %xh)))
     (lambda ()
      (receive r ((current-exception-handler) exn)
        (if (uncontinuable-exception? exn)
          (raise (make-error "returned from uncontinuable exception")(apply values r)))
          (lambda () (set! %xh prev))))))

(define (with-exception-handler handler thunk)
  (let ((prev %xh))
    (dynamic-wind
     (lambda () (set! %xh (cons handler %xh)))
     thunk
     (lambda () (set! %xh prev)))))

6.19.4 Conditions

Built-in Condition classes

Gauche currently has the following hierarchy of built-in condition classes. It approximately reflects SRFI-35 and SRFI-36 condition hierarchy, although they have Gauche-style class names. If there’s a corresponding SRFI condition type, the class has the SRFI name as well.

<condition>
  +- <compound-condition>
  +- <serious-condition>
    |   +- <serious-compound-condition> ; also inherits <compound-condition>
    +- <message-condition>
      +- <error> ; also inherits <serious-condition>
        +- <system-error>
        +- <unhandled-signal-error>
        +- <continuation-violation>
        +- <read-error>
        +- <io-error>
          +- <port-error>
```
Chapter 6: Core library

Note that some conditions may occur simultaneously; for example, error during reading from a file because of device failure may consist both <system-error> and <io-read-error>. In such cases, a compound condition is raised. So you can’t just use, for instance, (is-a? obj <io-read-error>) to check if <io-read-error> is thrown. See the "Condition API" section below.

<condition-meta>
[Metaclass]
Every condition class is an instance of this class. This class defines object-apply so that you can use a condition class as a predicate, e.g.:

\[
(condition<error> obj) \equiv (condition-has-type? obj <error>)
\]

<condition>
[Class]
&condition
[Condition Type]

[SRFI-35] The root class of the condition hierarchy.

<compound-condition>
[Class]
Represents a compound condition. A compound condition can be created from one or more conditions by make-compound-condition. Don’t use this class directly.

A compound condition returns #t for condition-has-type? if any of the original conditions has the given type.

<serious-condition>
[Class]
&serious
[Condition Type]

[SRFI-35] Conditions of this class are for the situations that are too serious to ignore or continue. Particularly, you can safely assume that if you raise this type of condition, it never returns.

<serious-compound-condition>
[Class]
This is an internal class to represent a compound condition with any of its component condition is serious. Inherits both <compound-condition> and <serious-condition>. make-compound-condition uses this class if the passed conditions includes a serious one. Don’t use this class directly.

<message-condition>
[Class]
&message
[Condition Type]

[SRFI-35] This class represents a condition with a message. It has one slot.

message
[Instance Variable of <message-condition>]
A message.

<error>
[Class]
&error
[Condition Type]

[SRFI-35] Indicates an error. Inherits <serious-condition> and <message-condition>, thus has message slot.

Note: SRFI-35 &error condition only inherits &serious and not &message, so you have to use compound condition to attach a message to the error condition. Gauche uses multiple
inheritance here, largely because of backward compatibility. To write a portable code, an
error condition should be used with a message condition, like this:

```
(condition
  (&message (message "Error message"))
  (&error))
```

[Function] `message-condition? obj`  
[Function] `serious-condition? obj`  
[Function] `error? obj`  

[SRFI-35] These are SRFI-35 compatible predicates to see if `obj` is a condition of type
`&message`, `&serious`, and `&error`, respectively.

```<system-error>```
A subclass of `<error>`. When a system call returns an error, this type of exception is
thrown. The `message` slot usually contains the description of the error (like the one from
`strerror(3)`). Besides that, this class has one more instance slot:

```errno```
[Instance Variable of `<system-error>`]  
Contains an integer value of system’s error number.

Error numbers may differ among systems. Gauche defines constants for typical Unix error
values (e.g. `EACCES`, `EBADF`, etc), so it is desirable to use them instead of literal numbers.
See the description of `sys-strerror` in Section 6.24.8 [System inquiry], page 313, for
available constants.

This class doesn’t have corresponding SRFI condition type, but important to obtain OS’s
raw error code. In some cases, this type of condition is compounded with other condition
types, like `<io-read-error>`.

```<unhandled-signal-error>```
A subclass of `<error>`. The default handler of most of signals raises this condition. See
Section 6.24.7.3 [Handling signals], page 309, for the details.

```signal```
[Instance Variable of `<unhandled-signal-error>`]  
An integer indicating the received signal number. There are constants defined for typical
signal numbers; see Section 6.24.7.1 [Signals and signal sets], page 307.

```<read-error>```
[Class]  
[SRFI-36] A subclass of `<error>`. When the reader detects a lexical or syntactic error during
reading an S-expression, this type of condition is raised.

```port```
[Instance Variable of `<read-error>`]  
A port from which the reader is reading. (NB: SRFI-36’s `&read-error` doesn’t have this
slot. Portable program shouldn’t rely on this slot).

```line```
[Instance Variable of `<read-error>`]  
A line count (1-base) of the input where the reader raised this error. It may be -1 if the
reader is reading from a port that doesn’t keep track of line count.

```column```
[Instance Variable of `<read-error>`]  
```position```
[Instance Variable of `<read-error>`]  
```span```
[Instance Variable of `<read-error>`]  
These slots are defined in SRFI-36’s `&read-error`. For the time being, these slots always
hold #f.
<io-error> [Class] &io-error [Condition Type]

<port-error> [Class] &io-port-error [Condition Type]

    port [Instance Variable of <port-error>]
        Holds the port where the error occurred.

<io-read-error> [Class] &io-read-error [Condition Type]

<io-write-error> [Class] &io-write-error [Condition Type]

<io-closed-error> [Class] &io-closed-error [Condition Type]
    [SRFI-36] An I/O error when read/write is attempted on a closed port. Inherits <port-error>.

<io-unit-error> [Class] &io-unit-error [Condition Type]
    An I/O error when the read/write is requested with a unit that is not supported by the port (e.g. a binary I/O is requested on a character-only port). Inherits <port-error>.

<continuation-violation> [Class] &continuation [Condition Type]
    [SRFI-226] An error thrown on violation of requirements related to continuations. A typical case is that a requested continuation prompt tag doesn’t exist in the context. See Section 6.15.8 [Continuation prompts], page 227, for the details.

make-continuation-violation prompt-tag [Function]
    [SRFI-226] Creates a new continuation violation condition, with prompt-tag as the continuation prompt tag that caused the violation.

continuation-violation? obj [Function]
    [SRFI-226] Returns #t if obj is a <continuation-violation> condition, #f otherwise.

continuation-violation-prompt-tag condition [Function]
    [SRFI-226] Extract the value of prompt-tag slot from condition, which must be a <continuation-violation> condition.

Condition API

define-condition-type name supertype predicate field-spec ... [Macro]
    [SRFI-35+] Defines a new condition type. In Gauche, a condition type is a class, whose metaclass is <condition-meta>.

    Name becomes the name of the new type, and also the variable of that name is bound to the created condition type. Supertype is the name of the supertype (direct superclass) of this condition type. A condition type must inherit from <condition> or its descendants. (Multiple inheritance can’t be specified by this form, and generally should be avoided in condition type hierarchy. Instead, you can use compound conditions, which don’t introduce multiple inheritance.)
A variable predicate is bound to a predicate procedure for this condition type.

Each field-spec is a form of (field-name accessor-name), and the condition will have fields named by field-name, and a variable accessor-name will be bound to a procedure that accesses the field. In Gauche, each field becomes a slot of the created class.

Gauche extends SRFI-35 to allow predicate and/or accessor-name to be #f, or accessor-name to be omitted, if you don’t need to them to be defined.

When define-condition-type is expanded into a class definition, each slot gets a :init-keyword slot option with the keyword whose name is the same as the slot name.

**condition-type? obj**  
[Function]  
[SRFI-35] Returns #t iff obj is a condition type. In Gauche, it means (is-a? obj <condition-meta>).

**make-condition-type name parent field-names**  
[Function]  
[SRFI-35] A procedural version to create a new condition type.

**make-condition type field-name value . . .**  
[Function]  
[SRFI-35] Creates a new condition of condition-type type, and initializes its fields as specified by field-name and value pairs.

**condition? obj**  
[Function]  
[SRFI-35] Returns #t iff obj is a condition. In Gauche, it means (is-a? obj <condition>).

**condition-has-type? obj type**  
[Function]  
[SRFI-35] Returns #t iff obj belongs to a condition type type. Because of compound conditions, this is not equivalent to is-a?.

**condition-ref condition field-name**  
[Function]  
[SRFI-35] Retrieves the value of field field-name of condition. If condition is a compound condition, you can access to the field of its original conditions; if more than one original condition have field-name, the first one passed to make-compound-condition has precedence.

You can use slot-ref and/or ref to access to the field of conditions; compound conditions define a slot-missing method so that slot-ref behaves as if the compound conditions have all the slots of the original conditions. Using condition-ref increases portability, though.

**condition-message condition :optional fallback**  
[Function]  
[SRFI-35+] This is a convenience procedure to retrieve a message if condition has a type <message-condition>, and returns fallback otherwise. If fallback is omitted, #f is assumed.

Often, in a generic routine you want to intercept a raised condition and retrieve a message for logging or user feedback. Since any object can be raised in Scheme, an exception may not contain the message slot, and you need to check the type of the condition. We’ve written such code enough so that we add this procedure.

**make-compound-condition condition0 condition1 . . .**  
[Function]  
[SRFI-35] Returns a compound condition that has all condition0 condition1 . . .. The returned condition’s fields are the union of all the fields of given conditions; if any conditions have the same name of fields, the first one takes precedence. The returned condition also has condition-type of all the types of given conditions. (This is not a multiple inheritance. See <compound-condition> above.)

**extract-condition condition condition-type**  
[Function]  
[SRFI-35] Condition must be a condition and have type condition-type. This procedure returns a condition of condition-type, with field values extracted from condition.
condition type-field-binding ...

[SRFI-35] A convenience macro to create a (possibly compound) condition. Type-field-binding is a form of (condition-type (field-name value-expr) ...).

(condition
  (type0 (field00 value00) ...)
  (type1 (field10 value10) ...)
  ...)
≡
(make-compound-condition
  (make-condition type0 'field00 value00 ...)  
  (make-condition type1 'field10 value10 ...)  
  ...)

6.20 Eval and repl

eval expr env

[R7RS eval] Evaluate expr under the environment env. In Gauche, env is just a <module> object.

R5RS and R7RS provide a portable way to obtain environment. R5RS way is described below. R7RS way is described in Section 10.2.7 [R7RS eval], page 588.

null-environment version

scheme-report-environment version

interaction-environment

[R5RS] Returns an environment specifier which can be used as the second argument of eval. Right now an environment specifier is just a module. (null-environment 5) returns a null module, which contains just the syntactic bindings specified in R5RS, (scheme-report-environment 5) returns a scheme module, which contains syntactic and procedure bindings in R5RS, and (interaction-environment) returns a user module that contains all the Gauche built-ins plus whatever the user defined. It is possible that the Gauche adopts a first-class environment object in future, so do not rely on the fact that the environment specifier is just a module.

An error is signaled if a value other than 5 is passed as version argument.

read-eval-print-loop :optional reader evaluator printer prompter

This exports Gosh's default read-eval-print loop to applications. Each argument can be #f, which indicates it to use Gauche's default procedure(s), or a procedure that satisfies the following conditions.

reader A procedure that takes no arguments. It is supposed to read an expression and returns it.

evaluator A procedure that takes two arguments, an expression and an environment specifier. It is supposed to evaluate the expression and returns zero or more value(s).

printer A procedure that takes zero or more arguments. It is supposed to print out these values. The result of this procedure is discarded.

prompter A procedure that takes no arguments. It is supposed to print out the prompt. The result of this procedure is discarded.

Given those procedures, read-eval-print-loop runs as follows:
1. Prints the prompt by calling prompter.
2. Reads an expression by calling reader. If it returns EOF, exits the loop and returns from read-eval-print-loop.
3. Evaluates an expression by calling evaluator
4. Prints the result by calling printer, then repeats from 1.

When an error is signaled from one of those procedures, it is captured and reported by the default escape handler, then the loop restarts from 1.

It is OK to capture a continuation within those procedures and re-invoke them afterwards.

6.21 Input and Output

6.21.1 Ports

A port class. A port is Scheme’s way of abstraction of I/O channel. Gauche extends a port in number of ways so that it can be used in wide range of applications.

Textual and binary I/O

R7RS defines textual and binary ports. In Gauche, most ports can mix both text I/O and binary I/O. It is cleaner to think the two is distinct, for they are sources/sinks of different types of objects and you don’t need to mix textual and binary I/O.

In practice, however, a port is often a tap to an untyped pool of bytes and you may want to decide interpret it later. One example is the standard I/O; in Unix-like environment, it’s up to the program to use pre-opened ports for textual or binary I/O. R7RS defines the initial ports for current-input-port etc. are textual ports; in Gauche, you can use either way.

Conversion

Some ports can be used to convert a data stream from one format to another; one of such applications is character code conversion ports, provided by gauche.charconv module (see Section 9.5 [Character code conversion], page 399, for details). SRFI-181 also provides this feature (see Section 11.40.2 [Transcoded ports], page 761).

Extra features

There are also a ports with special functionality. A coding-aware port (see Section 6.21.6 [Coding-aware ports], page 270) recognizes a special "magic comment" in the file to know which character encoding the file is written. Virtual ports (see Section 9.40 [Virtual ports], page 572) allows you to program the behavior of the port in Scheme. It is also provided by SRFI-181 (see Section 11.40 [Custom ports], page 760).

Ports can be combined to provide communication channel. See Section 12.9 [Plumbing ports], page 819, for utilities to do so.

6.21.2 Port and threads

When Gauche is compiled with thread support, the builtin port operations locks the port, so that port access from multiple threads will be serialized. (It is required by SRFI-18, BTW). Here, "builtin port operations" are the port access functions that takes a port and does some I/O or query on it, such as read/write, read-char/write-char, port->string, etc. Note that call-with* and with-* procedures do not lock the port during calling the given procedures, since the procedure may pass the reference of the port to the other thread, and Gauche wouldn’t know if that’s the case.
This means you don’t need to be too paranoia to worry about ports under multithreaded environment. However, keep it in mind that this locking mechanism is meant to be a safety net from breaking the port’s internal state, and not to be a general mutex mechanism. It assumes port accesses rarely conflict, and uses spin lock to reduce the overhead of majority cases. If you know there will be more than one thread accessing the same port, you should use explicit mutex to avoid conflicts.

with-port-locking port thunk

Executes thunk, while making the calling thread hold the exclusive lock of port during the dynamic extent of thunk.

Calls of the builtin port functions during the lock is held would bypass mutex operations and yield better performance.

Note that the lock is held during the dynamic extent of thunk; so, if thunk invokes a continuation captured outside of with-port-locking, the lock is released. If the continuation captured within thunk is invoked afterwards, the lock is re-acquired.

With-port-locking may be nested. The lock is valid during the outermost call of with-port-locking.

Note that this procedure uses the port’s built-in lock mechanism which uses busy wait when port access conflicts. It should be used only for avoiding fine-grain lock overhead; use explicit mutex if you know there will be conflicts.

6.21.3 Common port operations

port? obj

input-port? obj

output-port? obj

[R7RS base] Returns true if obj is a port, an input port and an output port, respectively.

Port? is not listed in the R5RS standard procedures, but mentioned in the "Disjointness of Types" section.

port-closed? port

Returns true if obj is a port and it is already closed. A closed port can’t be reopened.

current-input-port

current-output-port

current-error-port

[R7RS base] Returns the current input, output and error output port, respectively.

R7RS defines that the initial values of these ports are textual ports. In Gauche, initial ports can handle both textual and binary I/O.

Values of the current ports can be temporarily changed by parameterize (see Section 6.16.1 [Parameters], page 233), though you might want the convenience procedures such as with-output-to-string or with-input-from-file in typical cases.

(let1 os (open-output-string)
 (parameterize ((current-output-port os))
 (display "foo")
 (get-output-string os))
⇒ "foo"

current-trace-port

A paremeter that holds an output port which debug trace goes to. The initial value is the same as the initial value of current-error-port.

The debug-print feature (see Section 6.25.1 [Debugging aid], page 326) and the macro trace feature (see Section 5.6.1 [Tracing macro expansion], page 100) uses this port.
standard-input-port [Parameter]
standard-output-port [Parameter]
standard-error-port [Parameter]

Returns standard i/o ports at the time the program started. These ports are the default values of current-input-port, current-output-port and current-error-port, respectively.

You can also change value of these procedures by parameterize, but note that (1) current-*-ports are initialized before the program execution, so changing values of standard-*-port won’t affect them, and (2) changing values these procedures only affect Scheme-world, and does not change system-level stdio file descriptors low-level libraries referring.

with-input-from-port port thunk [Function]
with-output-to-port port thunk [Function]
with-error-to-port port thunk [Function]

Calls thunk. During evaluation of thunk, the current input port, current output port and current error port are set to port, respectively. Note that port won’t be closed after thunk is executed.

with-ports iport oport eport thunk [Function]

Does the above three functions at once. Calls thunk while the current input, output, and error ports are set to iport, oport, and eport, respectively. You may pass #f to any port argument(s) if you don’t need to alter the port(s).

Note that port won’t be closed after thunk is executed. (Unfortunately, recent Scheme standards added a similar named procedure, call-with-port, which does close the port. See below.)

close-port port [Function]
close-input-port port [Function]
close-output-port port [Function]

[R7RS base] Closes the port. Close-port works both input and output ports, while close-input-port and close-output-port work only for the respective ports and throws an error if another type of port is passed.

Theoretically, only close-port would suffice; having those three is merely for historical reason. R5RS has close-input-port and close-output-port; R6RS and R7RS support all three.

call-with-port port proc [Function]

[R7RS base] Calls proc with one argument, port. After proc returns, or it throws an un-captured error, port is closed. Value(s) returned from proc will be the return value(s) of call-with-port.

port-type port [Function]

Returns the type of port in one of the symbols file, string or proc.

port-name port [Function]

Returns the name of port. If the port is associated to a file, it is the name of the file. Otherwise, it is some description of the port.

port-buffering port [Function]
(setter port-buffering) port buffering-mode [Function]

If port is type of file port (i.e. (port-type port) returns file), these procedures gets and sets the port’s buffering mode. For input ports, the port buffering mode may be either one of :full, :modest or :none. For output ports, port-buffering, it may be one of :full, :line or :none. See Section 6.21.4 [File ports], page 264, for explanation of those modes.
If `port-buffering` is applied to ports other than file ports, it returns `#f`. If the setter of `port-buffering` is applied to ports other than file ports, it signals an error.

`port-link! input-port output-port`  
`port-unlink! port`  
An input port and an output port can be linked. After being linked, attempt to read from `input-port` causes flushing `output-port`'s buffer. A port cannot be linked to more than one port.

By default, standard input port is linked to standard output port.

The `port-link!` procedure links the given input port and output port. Both port must not be already linked, or an error is signaled.

The `port-unlink!` procedure removes the link between the given port and another port, if there's a link. If the given port is not linked, it does nothing.

You can use `link` slot of the port to see if a port is linked or not, and if linked, to which port.

`port-current-line port`  
Returns the current line count of `port`. This information is only available on file-based port, and as long as you're doing sequential character I/O on it. Otherwise, this returns -1.

`port-file-number port :optional dup?`  
Returns an integer file descriptor, if the `port` is associated to the system file I/O. Returns `#f` otherwise.

If a true value is passed to `dup?`, the procedure calls `dup(2)` and returns a duplicated file descriptor. In such case, the returned file descriptor can be closed independently with the port, and it is caller's responsibility to close it (by `sys-close`) once it's done.

`port-position port`  
|SRFI-192| Returns the current position of `port`. For the input port, the current position is where the next data will be read from, and for the output port, it is where the next data will be written to. (Note that `peek-char/peek-byte` won't move the current position).

If `port` is a simple file port or a string port, the position is represented as an integer byte offset from the top of the stream. If it is more complicated ports, such as the one doing the character encoding conversion, the port position may not be available, or returned object may not correspond to the byte offset. If `port` is a procedural (virtual) port (see Section 9.40 [Virtual ports], page 572), the returned value can be an arbitrary Scheme object, it can only be valid to be used for `set-port-position!`.

An error is thrown if `port` doesn't support port positions. You can use `port-has-port-position?` to check if the port supports port positions.

For portable code: SRFI-192 defines the following conditions.

- Return value is an arbitrary Scheme object and you should treat it as an opaque object. In general, it is only safe to pass it to the `set-port-position!` on the same port.
- If the port is a binary port and the returned position is a nonnegative exact integer, it represents the byte offset of the position.

`port-has-port-position? port`  
|SRFI-192| Returns `#t` iff the port supports `port-position` procedure.  

`set-port-position! port pos`  
|SRFI-192| Sets the current position of `port` to `pos`. For the input port, the current position is where the next data will be read from, and for the output port, it is where the next data will be written to.
Interpretation of pos is up to the port, and generally it must be an object returned previously by port-position on the same port. If port is a simple file port (a port opened on file, without CES conversion) or an input string port, you may pass a nonnegative exact integer as the byte offset.

An error is signaled if the position is not settable, or pos is not acceptable as a position for port. You can check if position of port is settable by port-has-set-port-position!?.

port-has-set-port-position!? port
[SRFI-192] Returns #t if the port supports set-port-position! procedure.

port-seek port offset :optional whence
(This procedure is deprecated. Use port-position and set-port-position! for the new code.
If the given port allows random access, this procedure sets the read/write pointer of the port according to the given offset and whence, then returns the updated offset (number of bytes from the beginning of the data). If port is not random-accessible, #f is returned. In the current version, file ports and input string ports are fully random-accessible. You can only query the current byte offset of output string ports.
Note that port position is represented by byte count, not character count.
It is allowed to seek after the data if port is an output file port. See POSIX lseek(2) document for details of the behavior. For input file port and input string port, you can’t seek after the data.
The whence argument must be a small integer that represents from where offset should be counted. The following constant values are defined.
SEEK_SET Offset represents the byte count from the beginning of the data. This is the default behavior when whence is omitted.
SEEK_CUR Offset represents the byte count relative to the current read/write pointer. If you pass 0 to offset, you can get the current port position without changing it.
SEEK_END Offset represents the byte count relative to the end of the data.

port-tell port
Returns the current read/write pointer of port in byte count, if port is random-accessible. Returns #f otherwise. This is equivalent to the following call:
(port-seek port 0 SEEK_CUR)

Note on the names: Port-seek is called seek, file-position or input-port-position/output-port-position on some implementations. Port-tell is called tell, ftell or set-file-position!. Some implementations have port-position for different functionality. Common Lisp has file-position, but it is not suitable for us since port need not be a file port. Seek and tell reflects POSIX name, and with Gauche naming convention we could use sys-seek and sys-tell; however, port deals with higher level of abstraction than system calls, so I dropped those names, and adopted new names.

copy-port src dst :key (unit 0) (size #f)
Copies data from an input port src to an output port dst, until eof is read from src.
The keyword argument unit may be zero, a positive exact integer, a symbol byte or a symbol char, to specify the unit of copying. If it is an integer, a buffer of the size (in case of zero, a system default size) is used to copy, using block I/O. Generally it is the fastest if you copy between normal files. If unit is a symbol byte, the copying is done byte by byte, using C-version of read-byte and write-byte. If unit is a symbol char, the copying is done character by character, using C-version of read-char and write-char.
If nonnegative integer is given to the keyword argument \texttt{size}, it specifies the maximum amount of data to be copied. If \texttt{unit} is a symbol \texttt{char}, \texttt{size} specifies the number of characters. Otherwise, \texttt{size} specifies the number of bytes.

Returns number of characters copied when \texttt{unit} is a symbol \texttt{char}. Otherwise, returns number of bytes copied.

### 6.21.4 File ports

#### 6.21.4.1 Opening file ports

**open-input-file** \texttt{filename :key if-does-not-exist buffering element-type} \hspace{1cm} [Function]

**open-output-file** \texttt{filename :key if-does-not-exist if-exists buffering} \hspace{1cm} [Function]

[R7RS+] Opens a file \texttt{filename} for input or output, and returns an input or output port associated with it, respectively.

The keyword arguments specify precise behavior.

**:if-exists**

This keyword argument can be specified only for \texttt{open-output-file}, and specifies the action when the \texttt{filename} already exists. One of the following value can be given.

**:supersede**

The existing file is truncated. This is the default behavior.

**:append**

The output data will be appended to the existing file.

**:overwrite**

The output data will overwrite the existing content. If the output data is shorter than the existing file, the rest of existing file remains.

**:error**

An error is signaled.

#:f

No action is taken, and the function returns #:f.

**:if-does-not-exist**

This keyword argument specifies the action when \texttt{filename} does not exist.

**:error**

An error is signaled. This is the default behavior of \texttt{open-input-file}.

**:create**

A file is created. This is the default behavior of \texttt{open-output-file}. The check of file existence and creation is done atomically; you can exclusively create the file by specifying :error or #:f to if-exists, along this option. You can’t specify this value for \texttt{open-input-file}.

#:f

No action is taken, and the function returns #:f.

**:buffering**

This argument specifies the buffering mode. The following values are allowed. The port’s buffering mode can be get/set by \texttt{port-buffering}. (see Section 6.21.3 [Common port operations], page 260).

**:full**

Buffer the data as much as possible. This is the default mode.

**:none**

No buffering is done. Every time the data is written (to an output port) or read (from an input port), the underlying system call is used. Process’s standard error port is opened in this mode by default.
This is valid only for output ports. The written data is buffered, but the buffer is flushed whenever a newline character is written. This is suitable for interactive output port. Process’s standard output port is opened in this mode by default. (Note that this differs from the line buffering mode of C stdio, which flushes the buffer as well when input is requested from the same file descriptor.)

This is valid only for input ports. This is almost the same as the mode :full, except that read-uvector may return less data than requested if the requested amount of data is not immediately available. (In the :full mode, read-uvector waits the entire data to be read). This is suitable for the port connected to a pipe or network.

This argument specifies the type of the file.

The file is opened in "binary" mode. (This is the default)

The file is opened in "character" (or "text") mode.

Note: This flag makes difference only on Windows-native platforms, and only affect the treatment of line terminators. In character mode, writing \newline on the output causes CR + LF characters to be written, instead of just LF. And reading CR + LF sequence returns just \newline.

On Unix, both mode are the same.

Note that Gauche doesn’t distinguish character (textual) port and binary port. So this flag really matters only on Windows line terminators.

This argument specifies character encoding of the file. The argument is a string or a symbol that names a character encoding scheme (CES).

For open-input-file, it can be a wildcard CES (e.g. *jp) to guess the file’s encoding heuristically (see Section 9.5.2 [Autodetecting the encoding scheme], page 402), or #t, in which case we assume the input file itself has magic encoding comment and use open-coding-aware-port (see Section 6.21.6 [Coding-aware ports], page 270).

If this argument is given, Gauche automatically loads gauche.charconv module and converts the input/output characters as you read to or write from the port. See Section 9.5.1 [Supported character encoding schemes], page 399, for the details of character encoding schemes.

If this argument is omitted, the value of the parameter default-file-encoding is assumed for the file’s character encoding (see Section 6.21.4.2 [Character encoding of file I/O], page 267).

This argument may be used with the encoding argument to specify the buffer size of character encoding conversion. It is passed as a buffer-size argument of the conversion port constructors (see Section 9.5.3 [Conversion ports], page 402).

Usually you don’t need to give this argument; but if you need to guess the input file encoding, larger buffer size may work better since guessing routine can have more data before deciding the encoding.

This argument may be used with the encoding argument to specify the behavior when the source character can’t be mapped to the destination character. It must
be either a symbol \texttt{raise} or a symbol \texttt{replace}, and is passed as \texttt{illegal-output} argument to the conversion port. See Section 9.5.3 [Conversion ports], page 402, for the details.

By combination of \texttt{if-exists} and \texttt{if-does-not-exist} flags, you can implement various actions:

\begin{verbatim}
(open-output-file "foo" :if-exists :error)
⇒ ;opens "foo" exclusively, or error
\end{verbatim}

\begin{verbatim}
(open-output-file "foo" :if-exists #f)
⇒ ;opens "foo" exclusively, or returns #f
\end{verbatim}

\begin{verbatim}
(open-output-file "foo" :if-exists :append
    :if-does-not-exist :error)
⇒ ;opens "foo" for append only if it already exists
\end{verbatim}

To check the existence of a file without opening it, use \texttt{sys-access} or \texttt{file-exists?} (see Section 6.24.4.4 [File stats], page 301).

Note for portability: Some Scheme implementations (e.g. STk) allows you to specify a command to \texttt{filename} and reads from, or writes to, the subprocess standard input/output. Some other scripting languages (e.g. Perl) have similar features. In Gauche, \texttt{open-input-file} and \texttt{open-output-file} strictly operates on files (what the underlying OS thinks as files). However, you can use “process ports” to invoke other command in a subprocess and to communicate it. See Section 9.27.4 [Process ports], page 498, for details.

\begin{verbatim}
call-with-input-file string proc :key if-does-not-exist buffering
    element-type encoding conversion-buffer-size
[Function]
call-with-output-file string proc :key if-does-not-exist if-exists
    buffering element-type encoding conversion-buffer-size
[Function]
\end{verbatim}

[R7RS+] Opens a file specified by \texttt{string} for input/output, and call \texttt{proc} with one argument, the file port. When \texttt{proc} returns, or an error is signaled from \texttt{proc} that is not captured within \texttt{proc}, the file is closed.

The keyword arguments have the same meanings of \texttt{open-input-file} and \texttt{open-output-file}’s. Note that if you specify \texttt{#f} to \texttt{if-exists} and/or \texttt{if-does-not-exist}, \texttt{proc} may receive \texttt{#f} instead of a port object when the file is not opened.

Returns the value(s) \texttt{proc} returned.

\begin{verbatim}
with-input-from-file string thunk :key if-does-not-exist buffering
    element-type encoding conversion-buffer-size
[Function]
with-output-to-file string thunk :key if-does-not-exist if-exists
    buffering element-type encoding conversion-buffer-size
[Function]
\end{verbatim}

[R7RS file] Opens a file specified by \texttt{string} for input or output and makes the opened port as the current input or output port, then calls \texttt{thunk}. The file is closed when \texttt{thunk} returns or an error is signaled from \texttt{thunk} that is not captured within \texttt{thunk}.

Returns the value(s) \texttt{thunk} returns.

The keyword arguments have the same meanings of \texttt{open-input-file} and \texttt{open-output-file}’s, except that when \texttt{#f} is given to \texttt{if-exists} and \texttt{if-does-not-exist} and the opening port is failed, \texttt{thunk} isn’t called at all and \texttt{#f} is returned as the result of \texttt{with-input-from-file} and \texttt{with-output-to-file}.

\textbf{Notes on semantics of closing file ports:} R7RS states, in the description of \texttt{call-with-port} et al., that "If \texttt{proc} does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for read or write operation."
Gauche’s implementation slightly misses this criteria; the mere fact that an uncaptured error is thrown in proc does not prove the port will never be used. Nevertheless, it is very difficult to think the situation that you can do meaningful operation on the port after such an error is signaled; you’d have no idea what kind of state the port is in. In practical programs, you should capture error explicitly inside proc if you still want to do some meaningful operation with the port.

Note that if a continuation captured outside call-with-input-file et al. is invoked inside proc, the port is not closed. It is possible that the control returns later into the proc, if a continuation is captured in it (e.g. coroutines). The low-level exceptions (see Section 6.19.3.3 [Low-level exception handling mechanism], page 252) also doesn’t ensure closing the port.

### 6.21.4.2 Character encoding of file I/O

**default-file-encoding**  
A character encoding name assumed when you omit encoding keyword arguments in file port opening procedures, such as open-input-port, call-with-output-file, etc.

The default value is Gauche’s native encoding, returned by gauche-character-encoding (see Section 6.9 [Characters], page 160).

In most circumstances, you don’t need to change the value of this parameter. These days, utf-8 became de facto in data exchange, so if you compile Gauche with utf-8 internal encoding (default), it is recommended that you specify encoding argument explicitly only when you know you’re dealing with non-utf8 data.

However, suppose you have existing system that assumes, say, latin-1 encoding, and you’ve been using Gauche with none CES encoding (which means Gauche just treats an octet as a character). If you want to switch Gauche with utf8 internal encoding, you need to check every file opening calls to make sure they have :encoding latin-1. That would be a lot of work.

You can change this parameter to latin-1 instead, and the exiting calls work as if :encoding latin-1 is attached unless it already has explicit :encoding.

Note that source code is loaded using coding aware ports (see Section 6.21.6 [Coding-aware ports], page 270), with the default encoding being Gauche’s native encoding. The value of this parameter doesn’t have effect. Use encoding magic comment to specify the encoding of source code.

### 6.21.4.3 File descriptor ports

**open-input-fd-port**  
Creates and returns an input or output port on top of the given file descriptor. Buffering specifies the buffering mode as described in open-input-file entry above; the default is :full. Name is used for the created port’s name and returned by port-name.

A boolean flag owner? specifies whether fd should be closed when the port is closed. If it is #f, closing port doesn’t close fd. If it is #t, closing port automatically closes fd. It can also be a symbol dup, in which case fd is duplicated (by dup(2) system call) and the resulting port owns the new fd which will be closed automatically–but fd itself remains open, and can be closed independently.

**port-fd-dup!**  
Interface to the system call dup2(2). Atomically closes the file descriptor associated to toport, creates a copy of the file descriptor associated to fromport, and sets the new file descriptor to toport. Both toport and fromport must be file ports. Before the original file descriptor
of `toport` is closed, any buffered output (when `toport` is an output port) is flushed, and any buffered input (when `toport` is an input port) is discarded.

‘Copy’ means that, even the two file descriptors differ in their values, they both point to the same system’s open file table entry. For example they share the current file position; after `port-fd-dup!`, if you call `port-seek` on `fromport`, the change is also visible from `toport`, and vice versa. Note that this ‘sharing’ is in the system-level; if either `toport` or `fromport` is buffered, the buffered contents are not shared.

This procedure is mainly intended for programs that needs to control open file descriptors explicitly; e.g. a daemon process would want to redirect its I/O to a harmless device such as `/dev/null`, and a shell process would want to set up file descriptors before executing the child process.

### 6.21.5 String ports

String ports are the ports that you can read from or write to memory.

**open-input-string** `string :key name`  
[Function]  
[R7RS base][SRFI-6] Creates an input string port that has the content `string`. This is a more efficient way to access a string in order rather than using `string-ref` with incremental index.

```scheme
(define p (open-input-string "foo x"))
(read p) ⇒ foo
(read-char p) ⇒ #\space
(read-char p) ⇒ #\x
(read-char p) ⇒ #<eof>
(read-char p) ⇒ #<eof>
```

The `name` keyword argument is a Gauche extension. By default, the created port is named as `(input string port)`. It is mainly used for debugging. You can specify alternative name with this argument. As Gauche’s convention, file ports has the source file path as its name, so port names for debugging information should be parenthesized not to be taken as pathnames.

```scheme
gosh> (open-input-string "")
#<iport (input string port) 0x215c0c0>
gosh> (open-input-string "" :name "(user input)")
#<iport (user input) 0x22a4e40>
```

**get-remaining-input-string** `port`  
[Function]  
`Port` must be an input string port. Returns the remaining content of the input port. The internal pointer of `port` isn’t moved, so the subsequent read from `port` isn’t affected. If `port` has already reached to EOF, a null string is returned.

```scheme
(define p (open-input-string "abc\ndef"))
(read-line p) ⇒ "abc"
(get-remaining-input-string p) ⇒ "def"
(read-char p) ⇒ #\d
(read-line p) ⇒ "ef"
(get-remaining-input-string p) ⇒ ""
```

**open-output-string** `:key name`  
[Function]  
[R7RS base][SRFI-6] Creates an output string port. Anything written to the port is accumulated in the buffer, and can be obtained as a string by `get-output-string`. This is a far more efficient way to construct a string sequentially than pre-allocate a string and fill it with `string-set!`.

The `name` keyword argument is a Gauche extension. By default, the created port is named as `(output string port)`. It is mainly used for debugging. You can specify alternative name
with this argument. As Gauche’s convention, file ports has the source file path as its name, so port names for debugging information should be parenthesized not to be taken as pathnames.

```
gosh> (open-output-string)
#<oport (output string port) 0x22a4c00>
gosh> (open-output-string :name "(temporary output)"
#<oport (temporary output) 0x22a49c0>
```

**get-output-string** *port*  
[R7RS base][SRFI-6] Takes an output string port *port* and returns a string that has been accumulated to *port* so far. If a byte data has been written to the port, this function re-scans the buffer to see if it can consist a complete string; if not, an incomplete string is returned.

This doesn’t affect the *port’s* operation, so you can keep accumulating content to *port* after calling **get-output-string**.

**Function**

call-with-input-string  *string proc*  
call-with-output-string  *proc*  
with-input-from-string  *string thunk*  
with-output-to-string  *thunk*  

These utility functions are trivially defined as follows. The interface is parallel to the file port version.

```
(define (call-with-output-string proc)
  (let ((out (open-output-string)))
    (proc out)
    (get-output-string out)))

(define (call-with-input-string str proc)
  (let ((in (open-input-string str)))
    (proc in)))

(define (with-output-to-string thunk)
  (let ((out (open-output-string)))
    (with-output-to-port out thunk)
    (get-output-string out)))

(define (with-input-from-string str thunk)
  (with-input-from-port (open-input-string str) thunk))
```

**Function**

call-with-string-io  *str proc*  
with-string-io  *str thunk*  

```
(define (call-with-string-io str proc)
  (let ((out (open-output-string))
    (in (open-input-string str)))
    (proc in out)
    (get-output-string out)))

(define (with-string-io str thunk)
  (with-output-to-string
    (lambda ()
      (with-input-from-string str thunk))))
```
write-to-string obj :optional writer
read-from-string string :optional start end

These convenience functions cover common idioms using string ports.

\[(\text{write-to-string obj writer})\\equiv\\ (\text{with-output-to-string (lambda () (writer obj)))}\]

\[(\text{read-from-string string})\\equiv\\ (\text{with-input-from-string string read})\]

The default value of \textit{writer} is the procedure \textit{write}. The default values of \textit{start} and \textit{end} is 0 and the length of \textit{string}.

Portability note: Common Lisp has these functions, with different optional arguments. STk has \textit{read-from-string} without optional argument.

6.21.6 Coding-aware ports
A coding-aware port is a special type of procedural input port that is used by \textit{load} to read a program source. The port recognizes the magic comment to specify the character encoding of the program source, such as ;; -*- coding: utf-8 -*-, and makes an appropriate character encoding conversion. See Section 2.3 [Multibyte scripts], page 14, for the details of coding magic comment.

\textbf{open-coding-aware-port iport} [Function]
Takes an input port and returns an input coding aware port, which basically just pass through the data from \textit{iport} to its reader. However, if a magic comment appears within the first two lines of data from \textit{iport}, the coding aware port applies the necessary character encoding conversion to the rest of the data as they are read.

The passed port, \textit{iport}, is "owned" by the created coding-aware port. That is, when the coding-aware port is closed, \textit{iport} is also closed. The content read from \textit{iport} is buffered in the coding-aware port, so other code shouldn’t read from \textit{iport}.

By default, Gauche’s \textit{load} uses a coding aware port to read the program source, so that the coding magic comment works for the Gauche source programs (see Section 6.22.1 [Loading Scheme file], page 286). However, since the mechanism itself is independent from \textit{load}, you can use this port for other purposes; it is particularly useful to write a function that processes Scheme source programs which may have the coding magic comment.

6.21.7 Input
For the input-related procedures, the optional \textit{iport} argument must be an input port, and when omitted, the current input port is assumed.

6.21.7.1 Reading data
\textbf{read :optional iport} [Function]
[R7RS base] Reads an S-expression from \textit{iport} and returns it. Gauche recognizes the lexical structure specified in R7RS, and some additional lexical structures listed in Section 4.1 [Lexical structure], page 43.

If \textit{iport} has already reached to the end of file, an eof object is returned.

The procedure reads up to the last character that consists the S-expression, and leaves the rest in the port. It’s not like CommonLisp’s \textit{read}, which consumes whitespaces after S-expression by default.
read-with-shared-structure :optional iport
read/ss :optional iport

[SRFI-38] These procedures are defined in SRFI-38 to recognize shared substructure notation (#n=, #n#). Gauche’s builtin read recognizes the SRFI-38 notation, so these are just synonyms to read; these are only provided for SRFI-38 compatibility.

read-char :optional iport

[R7RS base] Reads one character from iport and returns it. If iport has already reached to the end, returns an eof object. If the byte stream in iport doesn’t consist a valid character, the behavior is undefined. (In future, a port will have a option to deal with invalid characters).

peek-char :optional iport

[R7RS base] Reads one character in iport and returns it, keeping the character in the port. If the byte stream in iport doesn’t consist a valid character, the behavior is undefined. (In future, a port will have a option to deal with invalid characters).

read-byte :optional iport
read-u8 :optional iport

[R7RS base] Reads one byte from an input port iport, and returns it as an integer in the range between 0 and 255. If iport has already reached EOF, an eof object is returned. This is traditionally called read-byte, and R7RS calls it read-u8. You can use either.

peek-byte :optional iport
peek-u8 :optional iport

[R7RS base] Peeks one byte at the head of an input port iport, and returns it as an integer in the range between 0 and 255. If iport has already reached EOF, an eof object is returned. This is traditionally called peek-byte, and R7RS calls it peek-u8. You can use either.

read-line :optional iport allow-byte-string?

[R7RS base] Reads one line (a sequence of characters terminated by newline or EOF) and returns a string. The terminating newline is not included. This function recognizes popular line terminators (LF only, CRLF, and CR only). If iport has already reached EOF, an eof object is returned.

If a byte sequence is read from iport which doesn’t constitute a valid character in the native encoding, read-line signals an error by default. However, if a true value is given to the argument allow-byte-string?, read-line returns a byte string (incomplete string) in such case, without reporting an error. It is particularly useful if you read from a source whose character encoding is not yet known; for example, to read XML document, you need to check the first line to see if there is a charset parameter so that you can then use an appropriate character conversion port. This optional argument is Gauche’s extension to R7RS.

read-string nchars :optional iport

[R7RS base] Read nchars characters, or as many characters as available before EOF, and returns a string that consists of those characters. If the input has already reached EOF, an eof object is returned.

consume-trailing-whitespaces :optional iport

Reads and discards consecutive whitespace characters up to (including) the first EOL, from iport. If iport is omitted, the current input port is used.

This is mainly for interactive REPL when the input is line buffered. Suppose you’re reading user’s input using read. If the input is line buffered, read won’t see any input until the user type RET. It means if the user type (read-line)RET, then read returns (read-line), and if you evaluate it immediately, it reads RET.

That is not users typically expect: the user wants to type the input for read-line. Hence, it is useful to consume trailing RET after reading the user’s initial input.
read-block nbytes :optional iport

This procedure is deprecated - use read-uvector instead (see Section 9.38.4 [Uvector block I/O], page 567).

Reads nbytes bytes from iport, and returns an incomplete string consisted by those bytes. The size of returned string may shorter than nbytes when iport doesn’t have enough bytes to fill. If nbytes is zero, a null string is always returned.

If iport has already reached EOF, an eof object is returned.

If iport is a file port, the behavior of read-block differs by the buffering mode of the port (See Section 6.21.4 [File ports], page 264, for the detail explanation of buffering modes).

- If the buffering mode is :full, read-block waits until nbytes data is read, except it reads EOF.
- If the buffering mode is :modest or :none, read-block returns shorter string than nbytes even if it doesn’t reach EOF, but the entire data is not available immediately.

If you want to write a chunk of bytes to a port, you can use either display if the data is in string, or write-uvector in gauche.uvector (see Section 9.38.4 [Uvector block I/O], page 567) if the data is in uniform vector.

eof-object

[R7RS base] Returns an EOF object.

eof-object? obj

[R7RS base] Returns true if obj is an EOF object.

char-ready? :optional port

[R7RS base] If a character is ready to be read from port, returns #t.

For now, this procedure actually checks only if next byte is immediately available from port. If the next byte is a part of a multibyte character, the attempt to read the whole character may block, even if char-ready? returns #t on the port. (It is unlikely to happen in usual situation, but theoretically it can. If you concern, use read-uvector to read the input as a byte sequence, then use input string port to read characters.)

byte-ready? :optional port

[u8-ready? :optional port]

[R7RS base] If one byte (octet) is ready to be read from port, returns #t.

This is traditionally called byte-ready?, and R7RS calls it u8-ready?. You can use either.

6.21.7.2 Reader lexical mode

reader-lexical-mode

Get/set the reader lexical mode. Changing this parameter switches behavior of the reader concerning some corner cases of the lexical syntax, where legacy Gauche syntax and R7RS syntax aren’t compatible.

In general, you don’t need to change this parameter directly. The lexical syntax matters at the read-time, while changing this parameter happens at the execution-time; unless you know the exact timing when each phase occurs, you might not get what you want.

The hash-bang directive #!gauche-legacy and #!r7rs indirectly affects this parameter; the first one sets the reader mode to legacy, and the second one to strict-r7.

The command-line argument -fwarn-legacy sets the default reader mode to warn-legacy. Change to this parameter during load is delimited within that load; once load is done, the value of this parameter is reset to the value when load is started.
The parameter takes one of the following symbols as a value:

**permissive**

This is the default mode. It tries to find a reasonable compromise between two syntax.

In string literals, hex escape sequence is first interpreted as R7RS lexical syntax. If the syntax doesn’t conform R7RS hex escape, it is interpreted as legacy Gauche hex escape syntax. For example, "\x30;a" is read as "0a", for the hex escape sequence including the terminating semicolon is read as R7RS hex escape sequence. It also reads "\x30a" as "0a", for the legacy Gauche hex escape always takes two hexadecimal digits without the terminator. With this mode, you can use R7RS hex escape syntax for the new code, and yet almost all legacy Gauche code can be read without a problem. However, if the legacy code has a semicolon followed by hex escape, it is interpreted as R7RS syntax and the incompatibility arises.

**strict-r7**

Strict R7RS compatible mode. When the reader encounters the hash-bang directive `#!r7rs`, the rest of file is read with this mode.

In this mode, Gauche’s extended lexical syntax will raise an error.

Use this mode to ensure the code can be read on other R7RS implementations.

**legacy**

The reader works as the legacy Gauche (version 0.9.3.3 and before). When the reader encounters the hash-bang directive `#!gauche-legacy`, the rest of file is read with this mode.

This only matters when you want to read two-digit hex escape followed by semicolon as a character plus a semicolon, e.g. "\x30a" as "0;a" instead of "0a". We expect such a sequence rarely appears in the code, but if you dump a data in a string literal format, you may have such sequence (especially in incomplete string literals).

**warn-legacy**

The reader works as the *permissive* mode, but warns if it reads legacy hex-escape syntax. This mode is default when `-fwarn-legacy` command-line argument is given to *gosh*.

This is useful to check if you have any incompatible escape sequence in your code.

### 6.21.7.3 Read-time constructor

Read-time constructor, defined in SRFI-10, provides an easy way to create an external representation of user-defined structures.

```
#, (tag arg ...) [Reader Syntax]  
[SRFI-10] Gauche maintains a global table that associates a tag (symbol) to a constructor procedure.

When the reader encounters this syntax, it reads arg ..., finds a reader constructor associated with tag, and calls the constructor with arg ... as arguments, then inserts the value returned by the constructor as the result of reading the syntax.

Note that this syntax is processed inside the reader—the evaluator doesn’t see any of args, but only sees the object the reader returns.

define-reader-ctor tag procedure [Function]  
```
Chapter 6: Core library 274

Examples:

```
(define-reader-ctor 'pi (lambda () (* (atan 1) 4)))

#,(pi) ⇒ 3.141592653589793

'(#,pi)) ⇒ (3.141592653589793)

(define-reader-ctor 'hash
  (lambda (type . pairs)
    (let ((tab (make-hash-table type)))
      (for-each (lambda (pair)
                 (hash-table-put! tab (car pair) (cdr pair)))
        pairs)
    tab)))

(define table
  #,(hash eq? (foo . bar) (duh . dah) (bum . bom)))

table ⇒ #<hash-table eq? 0x80f9398>

(hash-table-get table 'duh) ⇒ dah
```

Combined with write-object method (see Section 6.21.8 [Output], page 275), it is easy to make a user-defined class written in the form it can be read back:

```
(define-class <point> ()
  ((x :init-value 0 :init-keyword :x)
   (y :init-value 0 :init-keyword :y)))

(define-method write-object ((p <point>) out)
  (format out "#,(<point> ~s ~s)" (ref p 'x) (ref p 'y)))

(define-reader-ctor '<point>
  (lambda (x y) (make <point> :x x :y y)))
```

**NOTE:** The extent of the effect of define-reader-ctor is not specified in SRFI-10, and might pose a compatibility problem among implementations that support SRFI-10. (In fact, the very existence of define-reader-ctor is up to an implementation choice.)

In Gauche, at least for the time being, define-reader-ctor take effects as soon as the form is compiled and evaluated. Since Gauche compiles and evaluates each toplevel form in order, tag specified in define-reader-ctor can be used immediately after that. However, it doesn’t work if the call of define-reader-ctor and the use of tag is enclosed in a begin form, for the entire begin form is compiled at once before being evaluated.

Other implementations may require to read the entire file before making its define-reader-ctor call effective. If so, it effectively prevents one from using define-reader-ctor and the defined tag in the same file. It is desirable to separate the call of define-reader-ctor and the use of tag in the different files if possible.

Another issue about the current define-reader-ctor is that it modifies the global table of Gauche system, hence it is not modular. The code written by different people might use the same tags, and yield an unexpected result. In future versions, Gauche may have some way to encapsulate the scope of tag, although the author doesn’t have clear idea yet.

6.21.7.4 Input utility functions

port->string port  [Function]
port->list reader port
port->string-list port
port->sexp-list port

Generally useful input procedures. The API is taken from scsh and STk.

port->string reads port until EOF and returns the accumulated data as a string.

port->list applies reader on port repeatedly, until reader returns an EOF, then returns the list of objects reader returned. Note that port isn’t closed.

port->string-list is a port->list specialized by read-line, and port->sexp-list is a port->list specialized by read.

If the input contains an octet sequence that’s not form a valid character in the Gauche’s native character encoding, port->string and port->string-list may return incomplete string(s).

If you want to deal with binary data, consider using port->uvector in gauche.uvector (see Section 9.38.4 [Uvector block I/O], page 567).

port-fold fn knil reader
port-fold-right fn knil reader
port-for-each fn reader
port-map fn reader

Convenient iterators over the input read by reader.

Since these procedures are not really about ports, they are superseded by generator-fold, generator-fold-right, generator-for-each and generator-map, respectively. See Section 6.15.11 [Folding generated values], page 231, for the details.

We provide these only for the backward compatibility.

6.21.8 Output

6.21.8.1 Layers of output routines

Gauche has quite a few output procedures which may confuse newcomers. The following table will help to understand how to use those procedures:

Object writers

Procedures that write out Scheme objects. Although there exist more low-level procedures, these are regarded as a basic layer of output routines, since it works on a generic Scheme object as a single unit. They come in two flavors:

- Write-family procedures: write, write-shared, write-simple—these are to produce external representation of Scheme objects, which can be generally read back by read without losing information as much as possible\(^1\). The external representation of most Scheme objects are the ones you write literal data in program, so this is the default way of writing Scheme objects out.
- Display-family procedures: display, print, newline. These are to produce plain-text output suitable for human readers.

High-level formatting output

To produce output in specific width, alignment, etc: format. This corresponds to C’s printf.

\(^1\) In a sense, this is somewhat similar to what is called “serialization” or “marshalling” in other programming language; you can write out a generic Scheme object on disk or to wire, and read it to get an object equivalent to the original one. In Lisp-family languages, this is called read/write invariance and is a built-in feature. Note that some objects do not have this invariance in nature, so sometimes you need to make your own serializer/marsher.
Low-level type-specific output
Procedures that deal with raw data.
- To output a character or a byte: `write-char`, `write-byte`.
- To output a string or an array of binary data: `write-string`, `write-uvector`.
- To flush the output buffer: `flush`, `flush-all-ports`.

6.21.8.2 Output controls

<write-controls>
You can control several aspects of Lisp structure output via `<write-controls>` object.
The object output routines (e.g. `write`, `display`) and the high-level output routines (e.g. `format`) can take optional write-controls.
The following example may give you some ideas on what write controls can do:

```lisp
(write '(1 10 100 1000)
  (make-write-controls :base 16 :radix #t))
```

prints `(\#x1 \#xa \#x64 \#x3e8)

```lisp
(write (iota 100)
  (make-write-controls :length 5))
```

prints `(0 1 2 3 4 ...)

The `make-write-controls` procedure returns a write-controls object, which has the following slots (those slot names are taken from Common Lisp’s print control variables):

- **length**: If this slot has a nonnegative integer, it determines the maximum number of items displayed for lists and vectors (including uniform vectors). If the sequence has more elements than the limit, an ellipsis is printed in place. If this slot is `#f` (default), sequence will be written out fully.

- **level**: If this slot has a nonnegative integer, it determines the maximum depth of the structure (lists and vectors) to be displayed. If the structure has deeper node, it will be printed as `. If this slot is `#f` (default), no depth limit is enforced.

- **base**: This slot must have an integer between 2 and 36, inclusive, and specifies the radix used to print exact numbers. The default value is 10.

- **radix**: This slot must have a boolean value. If it is true, radix prefix is always printed before exact numbers. The default value is `#f`.

- **pretty**: If this slot has true value, `pretty printing` is used, that is, newlines and indentations are inserted to show nested data structures fit in the specified width of columns.

- **width**: If this slot has a nonnegative integer, it specifies the display column width used for pretty printing.

- **indent**: This slot must be a nonnegative exact integer. If this is greater than zero, and the `pretty` slot is true, that number of whitespaces are written out after the pretty printer emits a
newline. That is, the second and after lines are indented. If the pretty slot is false, this slot is ignored and no indentation is done.

Note that the first line is not indended; the intention is that pretty printing begins at a certain column, so that no whitespace insertion is needed for the first line but the rest of lines need to be indented to align to the beginning of pretty print.

**string-length**

[Instance Variable of `<write-controls>`]

This slot must be a nonnegative exact integer or #f. If its value is an integer, string literals longer than the value are truncated during output; an ellipsis is shown after truncated content, before closing double-quotes.

A write-controls object is immutable. If you need a controls object with a slight variation of an existing controls object, use `write-controls-copy`.

Note: When we introduced `<write-controls>` object in 0.9.5, we used slot names as `print-length`, `print-pretty` etc., mirroring Common Lisp’s special variables. However, the `print-` part is redundant, as it is a part of a class dedicated to print control. So we changed the slot names as of 0.9.6. The procedures `make-write-controls` and `write-controls-copy` accepts both old and new names for the backward compatibility. The old code that directly refers to the slots needs to be rewritten (we think there’re a not a lot). We’ll drop the old name support in 1.0 release.

**make-write-controls**

`:key length level base radix pretty width indent`  
*String length*  

Creates and returns a write-controls object.

**write-controls-copy**

`controls :key length level base radix pretty width indent string-length`  
*Return a copy of another write-controls object controls. If keyword arguments are given, those values override the original values.*

Note: The high-level output procedures can be recursively called via `write-object` method. In that case, the write controls of the root output call will be automatically inherited to the recursive output calls to the same port.

**6.21.8.3 Object output**

For the following procedures, the optional port argument must be an output port, and when omitted, the current output port is assumed.

Some procedures take `port/controls` argument, which can be either an output port or `<write-controls>` object. For example, `write` takes up to two such optional arguments; that is, you can call it as `(write obj)`, `(write obj port)`, `(write obj controls)`, `(write obj port controls)` or `(write obj controls port)`. When omitted, the port is assumed to be the current output port, and the controls is assumed to be the default controls.

**write**

`obj :optional port/controls1 port/controls2`  
[Function]  

**write-shared**

`obj :optional port/controls1 port/controls2`  
[Function]  

**write-simple**

`obj :optional port/controls1 port/controls2`  
[Function]  

[R7RS+ write] The `write-family` procedures are used to write an external representation of Scheme object, which can be read back by `read` procedure. The three procedures differ in a way to handle shared or circular structures.

**Write** is circular-safe; that is, it uses datum label notation (#n= and #n#) to show cycles. It does not use datum label notation for non-circular structures that are merely shared (see the second example).

```
(let1 x (list 1)
```
(set-cdr! x x) ; create a cycle
(write x))
⇒ shows #0=(1 . #0#)

(let1 x (list 1)
  (write (list x x)))
⇒ shows ((1) (1))

Write-shared is also circular-safe, and it also shows shared structures using datum labels. Use this if you need to preserve topology of a graph structure.

(let1 x (list 1)
  (write-shared (list x x)))
⇒ shows (#0=(1) #0#)

Finally, write-simple writes out the object recursively without taking account of shared or circular structures. This is fast, for it doesn’t need to scan the structure before actually writing out. However, it won’t stop when a circular structure is passed.

When these procedures encounter an object of a user-defined class, they call the generic function write-object.

Historical context: Write has been in Scheme standards, but handling of circular structures hasn’t been specified until R7RS. In fact, until Gauche 0.9.4, write diverged for circular structures. SRFI-38 introduced the datum-label notation and write-with-shared-structure and write/ss procedures to produce such notation, and Gauche supported it. R7RS clarified this issue, and Gauche 0.9.4 followed.

write-with-shared-structure obj :optional port
[Function]
write/ss obj :optional port
[Function]
write* obj :optional port
[Function]

[SRFI-38] These are aliases of write-shared above.

Gauche has been used the name write* for long, which is taken from STklos. SRFI-38 defines write-with-shared-structure and write/ss. These names are kept for the backward compatibility. New code should use write-shared.

display obj :optional port/controls1 port/controls2
[Function]
[R7RS write] Produces a human-friendly representation of an object obj to the output port. If obj contains cycles, display uses datum-label notation.

When display encounters an object of a user-defined class, it calls the generic function write-object.

(display "\"Mahalo\", he said.")
⇒ shows "Mahalo", he said.

(let ((x (list "imua")))
  (set-cdr! x x)
  (display x))
⇒ shows #0=(imua . #0#)

print expr ...
[Function]
Displays exprs (using display) to the current output port, then writes a newline.

pprint obj :key port controls width length level newline indent
[Function]
Pretty prints obj to port, which is defaulted to the current output port. The same effect is achieved by passing the write procedure a write control with pretty slot setting to #t (in fact, it is how pprint is implemented), but this procedure provides more convenient interface when you want to play with the pretty printer.
By default, `pprint` prints a newline after writing `obj`. You can suppress this newline by passing `#f` to `newline` keyword argument.

To customize pretty printing, you can pass a write control object to the `controls` keyword argument (the `pretty` slot of `controls` is ignored; it’ll always printed prettily). Furthermore, you can override `width`, `length`, `level`, and `indent` slots of `controls`. If you omit `controls`, a reasonable default value is assumed. See Section 6.21.8.2 [Output controls], page 276, for the detail of write controls.

```
(pprint (make-list 6 '(gauche droite)))
⇒ prints
 ((gauche droite) (gauche droite) (gauche droite) (gauche droite)
 (gauche droite) (gauche droite))

(pprint (make-list 6 '(gauche droite)) :width 20)
⇒ prints
 ((gauche droite)
 (gauche droite)
 (gauche droite)
 (gauche droite)
 (gauche droite)
 (gauche droite))

(pprint (make-list 6 '(gauche droite)) :length 3)
⇒ prints
 ((gauche droite) (gauche droite) (gauche droite) ...)

(pprint (make-list 6 '(gauche droite)) :level 1)
⇒ prints
 (# # # # # #)
```

**write-object**

`write-object (obj <object>) port`

You can customize how the object is printed out by this method.

**newline**

`:optional port`

[R7RS base] Writes a newline character to `port`. This is equivalent to `(write-char \newline port)`, `(display "\n" port)`. It is kept for a historical reason.

**fresh-line**

`:optional port`

If the output column is at the beginning of line, do nothing and returns `#f`. Otherwise, writes a newline character and returns `#t`. It is handy when you want to make sure the next output starts from the beginning of line, but do not want to produce an extra blank line.

This procedure is taken from Common Lisp.

Not all situations allow this procedure to know the port’s output column; especially, if you mix textual and binary data to output, the column tracking becomes unreliable. If this procedure can’t determine the column position, it emits a newline character.

### 6.21.8.4 Formatting output

**format**

`string arg ...`

**format**

`dest string arg ...`

**format**

`controls string arg ...`

**format**

`dest controls string arg ...`
**format** controls dest string arg . . .

[Function]

[SRFI-28+] Format arg . . . according to string. This function is a subset of CommonLisp’s **format** function, with a bit of extension. It is also a superset of SRFI-28, Basic format strings.

The dest argument specifies the destination; if it is an output port, the formatted result is written to it; if it is #t, the result is written to the current output port; if it is #f, the formatted result is returned as a string. Dest can be omitted, as SRFI-28 **format**; it has the same effects as giving #f to the dest.

The controls argument is `<write-controls>` object (see Section 6.21.8.2 [Output controls], page 276), which affects the output of `~s` and `~a`. This is Gauche’s extension.

(The unusual function signature of **format** is for the convenience; both dest and controls are optional and they can appear in either order.)

`string` is a string that contains format directives. A format directive is a character sequence begins with tilde, `~`, and ends with some specific characters. A format directive takes the corresponding arg and formats it. The rest of string is copied to the output as is.

```
(format #f "the answer is ~s" 42)
⇒ "the answer is 42"
```

The format directive can take one or more parameters, separated by comma characters. A parameter may be an integer or a character; if it is a character, it should be preceded by a quote character. Parameter can be omitted, in such case the system default value is used.

The interpretation of the parameters depends on the format directive.

Furthermore, a format directive can take two additional flags: atmark ‘@’ and colon ‘:’. One or both of them may modify the behavior of the format directive. Those flags must be placed immediately before the directive character.

If a character ‘v’ or ‘V’ is in the place of the parameter, the value of the parameter is taken from the format’s argument. The argument must be either an integer, a character, or #f (indicating that the parameter is effectively omitted).

Some examples:

```
~10,2s  A format directive `~s`, with two parameters, 10 and 2.
~12,,,,**A  A format directive `~a`, with 12 for the first parameter and a character `*` for the fourth parameter. The second and third parameters are omitted.
~10@d  A format directive `~d`, with 10 for the first parameter and ‘@’ flag.
~v,vx  A format directive `~x`, whose first and second parameter will be taken from the arguments.
```

The following is a complete list of the supported format directives. Either upper case or lower case character can be used for the format directive; usually they have no distinction, except noted.

```
~A  Parameters: mincol,colinc,minpad,padchar,maxcol
Ascii output. The corresponding argument is printed by display. If an integer mincol is given, it specifies the minimum number of characters to be output; if the formatted result is shorter than mincol, a whitespace is padded to the right (i.e. the result is left justified).

The colinc, minpad and padchar parameters control, if given, further padding. A character padchar replaces the padding character for the whitespace. If an integer minpad is given and greater than 0, at least minpad padding character is used, regardless of the resulting width. If an integer colinc is given, the padding
```
character is added (after minpad) in chunk of colinc characters, until the entire width exceeds mincol.

If atmark-flag is given, the format result is right justified, i.e. padding is added to the left.

The maxcol parameter, if given, limits the maximum number of characters to be written. If the length of formatted string exceeds maxcol, only maxcol characters are written. If colon-flag is given as well and the length of formatted string exceeds maxcol, maxcol - 4 characters are written and a string “...” is attached after it.

```
(format #f "|~a|" "oops")
⇒ "|oops|
(format #f "|~10a|" "oops")
⇒ "|oops |
(format #f "|~10@a|" "oops")
⇒ "|********oops|
(format #f "|~,,,,,10a|" '(abc def ghi jkl))
⇒ "|(abc def gh|"
(format #f "|~,,,,,10:@a|" '(abc def ghi jkl))
⇒ "|(abc de ...|
```

~S Parameters: mincol, colinc, minpad, padchar, maxcol

S-expression output. The corresponding argument is printed by write. The semantics of parameters and flags are the same as ~A directive.

```
(format #f "|~s|" "oops")
⇒ "\"oops\"|
(format #f "|~10s|" "oops")
⇒ "\"oops\" |
(format #f "|~10@s|" "oops")
⇒ "| \"oops\"|
(format #f "|~10,,,'*@s|" "oops")
⇒ "|*****\"oops\"|
```

~W Parameters: None

Write prettily. Without flags, it just works like ~S. If a colon flag is given, it works like pretty print, i.e. pretty slot of the write context is true. If an atmark flag is given, it works like length and level slots of the write context is #f, i.e. temporarily turn off length and level limits.

The indent of pretty print is set at the position where ~W begins, so that if the output spans multiple lines, the second and following lines are indented properly.

```
(format #t "Result: ~:w\n" (make-list 40 'aaaa))
⇒ prints:
Result: (aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa)
```

~C Parameters: None

Character output. The argument must be a character, or an error is signaled. If no flags are given, the character is printed with display. If atmark-flag is given, the character is printed with write.
Parameters: mincol, padchar, commachar, interval

Decimal output. The argument is formatted as an decimal integer. If the argument is not an integer, all parameters are ignored (after processing ‘v’ parameters) and it is formatted by ‘A’ directive.

If an integer parameter mincol is given, it specifies minimum width of the formatted result; if the result is shorter than it, padchar is padded on the left (i.e. the result is right justified). The default of padchar is a whitespace.

\[
\text{format #f "|\-d\|" 12345} \\
\Rightarrow "|12345|"
\]

\[
\text{format #f "|\-10d\|" 12345} \\
\Rightarrow "| 12345|"
\]

\[
\text{format #f "|10,’0d\|" 12345} \\
\Rightarrow "|0000012345|"
\]

If atmark-flag is given, the sign ‘+’ is printed for the positive argument.

If colon-flag is given, every interval-th digit of the result is grouped and commachar is inserted between them. The default of commachar is ‘,’ and the default of interval is 3.

\[
\text{format #f "|\-:d\|" 12345} \\
\Rightarrow "|12,345|"
\]

\[
\text{format #f "|,,,\-,4:d\|" -12345678} \\
\Rightarrow "|\-1234_5678|"
\]

Parameters: mincol, padchar, commachar, interval

Binary output. The argument is formatted as a binary integer. The semantics of parameters and flags are the same as the ‘D’ directive.

Parameters: mincol, padchar, commachar, interval

Octal output. The argument is formatted as an octal integer. The semantics of parameters and flags are the same as the ‘D’ directive.

Parameters: mincol, padchar, commachar, interval

Hexadecimal output. The argument is formatted as a hexadecimal integer. If ‘X’ is used, upper case alphabets are used for the digits larger than 10. If ‘x’ is used, lower case alphabets are used. The semantics of parameters and flags are the same as the ‘D’ directive.

\[
\text{format #f "\-8,’0x" 259847592} \\
\Rightarrow "0f7cf5a8"
\]

\[
\text{format #f "\-8,’0X" 259847592} \\
\Rightarrow "0F7CF5A8"
\]

Parameters: radix, mincol, padchar, commachar, interval

Radix. If the argument is an exact number, the directive ‘nR prints out the argument in radix n. For example, ‘10R is the same as ‘D. The radix parameter must be between 2 and 36, inclusive. Other parameters are are the same as ‘D directive.

If the argument is other than an exact number, it is formatted just as ‘A.

If uppercase R is used, uppercase alphabets are used for base-11 and above. If lowercase r is used, lowercase alphabets are used.

If atmark-flag is given, it behaves differently; it prints out the numeric argument in Roman numerals, e.g. MMXXIII for 2023. Adding colon-flag with atmark-flag,
it uses a variation of Roman numerals (IIII for 4 instead of IV, for example). The argument must be a positive exact integer in this case. Common Lisp only supports up to 4,999 since the Roman numeral 5,000 can’t be written in ASCII. Unicode has Roman numeral up to 100,000, however, so we support up to 499,999. Again, the distinction of R and r is reflected to the ASCII alphabets used in the output.

If neither radix nor atmark-flag is given, Common Lisp prints the number in English. We haven’t supported it yet, and an error is raised.

\[
\begin{align*}
\text{format} \ #f \ "^-36R" \ 123456789 & \Rightarrow \ "21I3V9" \\
\text{format} \ #f \ "^-36r" \ 123456789 & \Rightarrow \ "21i3v9" \\
\text{format} \ #f \ "^@R" \ 1999 & \Rightarrow \ "MCMXCIX" \\
\text{format} \ #f \ "^@:R" \ 1999 & \Rightarrow \ "MDCCCLXXXXVIIII" \\
\text{format} \ #f \ "^@r" \ 1999 & \Rightarrow \ "mcmxcix" \\
\text{format} \ #f \ "^@R" \ 123456 & \Rightarrow \ "MMMCXLVI"
\end{align*}
\]

Parameters: \text{width,digis,\texttt{scale},ovfchar,padchar}

Floating-number output. If the argument is a real number, it is formatted as a decimal floating number. The \text{width} parameter defines the width of the field; the number is written out right-justified, with the left room padded with \text{padchar}, whose default is #\text{space}. When the formatted output can’t fit in \text{width}, \text{ovfchar} is output \text{width} times if it is given, or the entire output is shown if \text{ovfchar} is omitted.

\[
\begin{align*}
\text{format} \ "^-6f" \ 3.14 & \Rightarrow \ "3.14" \\
\text{format} \ "^-6f" \ 3.141592 & \Rightarrow \ "3.141592" \\
\text{format} \ "^-6,,,’f" \ 3.141592 & \Rightarrow \ "####" \\
\text{format} \ "^-6,,,’*f" \ 3.14 & \Rightarrow \ "**3.14"
\end{align*}
\]

The \text{digits} parameter specifies number of digits shown below the decimal point. Must be nonnegative integer. When omitted, enough digits to identify the flonum uniquely is generated (same as using \texttt{write} and \texttt{display}—when you read back the number, you’ll get exactly the same flonum.)

\[
\begin{align*}
\text{format} \ "^-6,3f" \ 3.141592 & \Rightarrow \ "3.142" \\
\text{format} \ "^-6,0f" \ 3.141592 & \Rightarrow \ "3.0" \\
\text{format} \ "^-10,4f" \ 355/113 & \Rightarrow \ "3.1416" \\
\text{format} \ "^-10,4f" \ 3 & \Rightarrow \ "3.0000"
\end{align*}
\]

If the \text{scale} parameter is given, the argument is multiplied by \texttt{(expt 10 scale)} before printing.

If the @ flag is given, plus sign is printed before the non-negative number.

\[
\text{format} \ "^-8,3@f" \ 3.141592 \Rightarrow \ "+3.142"
\]

When \text{digits} is smaller than the digits required to represent the flonum unambiguously, we round at \text{digits}+1 position. By default, it is done based on the value the flonum represents—that is, we choose the rounded value closer to the actual value of the flonum. It can sometimes lead to unintuitive results, however. Suppose you want to round 1.15 at 100ths (that is, round to nearest 10ths). Unlike elementary math class, it gives you 1.1. That’s because the flonum represented by 1.15 is actually tiny bit smaller than 1.15, so it’s closer to 1.1 than 1.2. We show it as 1.15 since no other flonums are closer to 1.15.

But in casual applications, users may perplexed with this behavior. So we support another rounding mode, which we call notational rounding. It is based on the notation used for the flonum. In that mode, rounding 1.15 to nearest 10ths yields 1.2. You can get it by adding : flag.
(format "-6,1f" 1.15) ⇒ " 1.1"
(format "-6,1:1f" 1.15) ⇒ " 1.2"

- $ Parameters: digits,idigits,width,padchar

Floating-point formatting suitable for currency display. The digits parameter specifies the number of digits after the decimal point (default 2). The idigits parameter specifies the minimum number of digits before the decimal point (default 1). The width parameter specifies the minimum number of characters for the entire display, and if the number of printed characters are smaller than it, padchar is displayed on the left to fill the blank. The default of width is 0, and the default of padchar is \space.

If @ flag is given and the argument is nonnegative, + is displayed.
If : flag is given, the sign is displayed first, before any padding characters.
If the argument isn’t a real number, the object is formatted as if ~wD directive is given (where w is width).
Gauche specific: The number is rounded with notational rounding (see the description of ~F above for the rounding mode).

(map (cut format "~$ <>") '(1.23 4.5 6)) ⇒ '("1.23" "4.50" "6.00")

- ? Parameters: None

Recursive formatting. The argument for this directive must be a string which is interpreted as a format string. The arguments for the given directive should be given in the next argument, as a list.

(format "~-s?-s" '< "~s ~s" '(a b) '>) ⇒ "<a b>"

If @ flag is given, the arguments for the given directive is taken from the following arguments.

(format "~-s@?-s" '< "~s ~s" 'a 'b '>) ⇒ "<a b>"

- * Parameter: count

Moves the argument counter count times forward, effectively skips next count arguments. The default value of count is 1, hence skip the next argument. If a colon-flag is given, moves the argument counter backwards, e.g. ~:* makes the next directive to process last argument again. If an atmark-flag is given, count specifies absolute position of the arguments, starting from 0.

- P Parameters: none

Plural. If argument is an exact integer 1, do nothing. Otherwise, print s. If an atmark-flag is given, print y for the argument 1, and print ies for other argument values.

If a colon-flag is given, back up one argument before processing.

Note: 1.0 is regarded as plural.

(format "-a book~:p, -a book~:p, -a pon~:0p, -a pon~:0p" 1 2 1 2) ⇒ "1 book, 2 books, 1 pony, 2 ponies"

- & Parameter: count

Fresh line. If the output position is at the beginning of line, do nothing; otherwise, output \newline, just like fresh-line procedure (see Section 6.21.8.3 [Object output], page 277).
If `count` is given and greater than 1, output `(- count 1)` newline characters after the fresh line operation. If `count` is 0, do nothing.

```
---
-\n
```

Parameters: `count`

Output a `#\n`, `#\newline`, `#\tab` and `#\page`, respectively. If `count` is given, output the specified number of characters.

```
- + newline
```

Parameters: none

Ignored newline. As Scheme supports backslash-newline escape in string literals, this is not necessary, but provided for Common Lisp compatibility. Without flags, the newline after the tilde, and any following whitespace characters (char-whitespace?) are ignored. With a colon flag, the newline immediately after the tilde is ignored but the following whitespace characters are preserved. With an atmark flag, the newline immediately after the tilde is kept but the following whitespace characters are ignored.

```
(format "abc\n  def") ⇒ "abc def"
(format "abc:\n  def") ⇒ "abc  def"
(format "abc@\n  def") ⇒ "abc\ndef"
(format "abc:\n  \ndef") ⇒ "abc  \ndef"
```

### 6.21.8.5 Low-level output

**write-char**  
*char :optional port*  
[R7RS base] Write a single character `char` to the output port `port`.

**write-byte**  
*byte :optional port*  
[R7RS base] Write a byte `byte` to the port. `byte` must be an exact integer in range between 0 and 255.

This is traditionally called `write-byte`, and R7RS calls it `write-u8`. You can use either.

**write-string**  
*string :optional oport start end*  
[R7RS base] If the optional `start` and `end` arguments are omitted, it is the same as `(display string oport)`. The optional arguments restricts the range of `string` to be written.

**flush**  
*optional port*  
[Function]  
Output the buffered data in `port`, or all ports, respectively.

R7RS's `flush-output-port` is the same as `flush`. The `scheme.base` module defines the name as an alias to `flush` (see Section 10.2.2 [R7RS base library], page 585).

The function "flush" is called in variety of ways on the various Scheme implementations: `force-output` (Scsh, SCM), `flush-output` (Gambit), or `flush-output-port` (Bigloo). The name `flush` is taken from STk and STKlos.

### 6.22 Loading Programs
6.22.1 Loading Scheme file

load file :key paths (error-if-not-found #t) environment ignore-coding       [Function]
[R7RS+] Loads file, that is, read Scheme expressions in file and evaluates them. An extension “.scm” may be omitted from file.
If file doesn’t begin with “/” or “.” or “../”, it is searched from the system file search list, stored in a variable *load-path*. Or you can explicitly specify the search path by passing a list of directory names to the keyword argument paths.
On success, load returns the string pathname of the file actually loaded. If the specified file is not found, an error is signaled unless the keyword argument error-if-not-found is #f, in which case load returns #f.
By default, load uses a coding-aware port (see Section 6.21.6 [Coding-aware ports], page 270) so that the "coding:" magic comment at the beginning of the source file is effective. (See Section 2.3 [Multibyte scripts], page 14, for the details of the coding magic comment). If a true value is given to the keyword argument ignore-coding, load doesn’t create the coding-aware port and directly reads from the file port.
If a module is given to the keyword argument environment, load works as if the given module is selected at the beginning of the loaded file.
The current module is preserved; even select-module is called in file, the module in which load is called is restored afterwards.
Gauche’s load is upper-compatible to R5RS load, but R7RS load differs in optional arguments; see Section 10.2.11 [R7RS load], page 590.
If you want to load a library file, it’s better to use ‘use’ (see Section 4.13.3 [Defining and selecting modules], page 81), or ‘require’ described below. See Section 2.7 [Compilation], page 17, for difference between load and require.

*load-path*       [Variable]
Keeps a list of directories that are searched by load and require.
If you want to add other directories to the search path, do not modify this variable directly; use add-load-path, described below, instead.

add-load-path path flag . . .       [Special Form]
Adds a path path to the library load path list. Path must be a literal string, for load paths must be known at compilation time. If path is a relative path, it is resolved relative to the current working directory, unless :relative flag is given.
Path doesn’t need to exist; nonexisting paths in load path list are simply ignored. However, if path does exist, add-load-path searches for architecture-dependent paths; see below.
Each flag argument may be one of the followings.
:after           Append path to the end of the current list of load paths. By default, path is added in front of the load path list.
#:t              The same as :after. This is for the backward compatibility.
:relative
Interpret path as a relative path to the directory of the current file, instead of the current working directory. If the current file can’t be determined (e.g. evaluated in REPL, or the expression is read from a socket), this flag is ignored.
Use this form instead of changing *load-path* directly. This form is a special form and recognized by the compiler; if you change *load-path*, it is in effect at run time, and that may be too late for “use” or “require”.
Furthermore, add-load-path looks for the architecture dependent directories under the specified path and if it exists, sets up the internal path list for dynamic loading correctly. Suppose you have your Scheme module in /home/yours/lib, and that requires a dynamic loadable library. You can put the library under /home/yours/lib/ARCH/, where ARCH is the value (gauche-architecture) returns (see Section 6.24.3 [Environment inquiry], page 295). Then you can have compiled libraries for multiple platforms and Gauche can still find the right library.

load-from-port port

[Function]
Reads Scheme expressions from an input port port and evaluates them, until EOF is read.

Note that unless you pass a coding-aware port to port, the "coding:" magic comment won’t be handled.

current-load-port

current-load-path

current-load-history

current-load-next

These procedures allows you to query the current context of loading. They returns the following values when called inside a file being loaded:

current-load-port
 Returns the port object from which this form is being loaded.

current-load-path
 Returns the pathname of the file from which this form is being loaded. Note that this may return #f if the source of load is not a file.

current-load-history
 Returns a list of pairs of a port and a line number (integer), representing the nesting of loads. Suppose you load foo.scm, and from its line 7 it loads bar.scm, and from its line 18 it loads baz.scm. If you call current-load-history in the file baz.scm, you’ll get

((<port "foo.scm"> . 7) (<port "bar.scm"> . 18))

current-load-next
 Returns a list of remaining directories to be searched at the time this file is found. Suppose the *load-path* is ("." "../lib" "/home/gauche/lib" "/share/gauche/lib") and you load foo.scm, which happens to be in ../lib/. Then, inside foo.scm, current-load-next returns:

("/home/gauche/lib" "/share/gauche/lib")

When called outside of load, these procedures returns #f, #f, () and (), respectively.

6.22.2 Load dynamic library

dynamic-load file :key init-function

[Function]
Loads and links a dynamic loadable library (shared library) file. File shouldn’t contain the suffix (".so" on most systems); dynamic-load adds it, for it may differ among platforms.

The keyword argument init-function specifies the initialization function name of the library in a string. By default, if the file basename (without extension) is “foo”, the initialization function name is “Scm_Init_foo”.

Usually a dynamic loadable library is provided with wrapping Scheme module, so the user doesn’t have to call this function directly.

There’s no way to unload the loaded libraries.
6.22.3 Require and provide

Require and provide are a traditional Lisp way to ensure loading a library file only once. If you require a feature for the first time, a library file that provides it is loaded and the fact that the feature is provided is memorized. Subsequent request of the same feature doesn’t need to load the file.

In Gauche, the use syntax (see Section 4.13.4 [Using modules], page 81) hides the require mechanism under the hood so you hardly need to see these forms. These are provided just in case if you want to do some non-trivial management of libraries and thus want to bypass Gauche’s standard mechanism.

**require** feature

If feature is not loaded, load it. Feature must be a string, and it is taken as a file name (without suffix) to be loaded. This loading takes place at compile time.

If you load SLIB module, require is extended. see Section 12.58 [SLIB], page 959, for details.

If the loaded file does not contain provide form at all, the feature is automatically provided, as if (provide feature) is called at the end of the loaded file. We call this autoprove feature.

Note that require first sets the current module to an immutable module called gauche.require-base and then load the file. The files loaded by require usually have define-module/select-module or define-library for the first thing, so you rarely notice the gauche.require-base module. However, if the loaded file has toplevel defines or imports (use’s) without specifying a module, you’ll get an error like the following:

```scheme
*** ERROR: Attempted to create a binding (a) in a sealed module: #<module gauche.require-base>
```

Rationale: Generally it’s difficult to guarantee when the specified file is loaded by require (because some other module may already have required it). If we just used the caller’s current module, there would be a couple of issues: The form define-module or define-library may not be visible from the current module, and you can’t guarantee if the toplevel defines without specifying modules in the loaded file inserts the caller’s current module, since they may have been loaded into a different module. It is just a bad idea to insert toplevel definitions or to import other modules without specifying which module you put them in. So we made them an error.

**provide** feature

Adds feature to the system’s provided feature list, so that the subsequent require won’t load the same file again.

Because of the autoproviding, i.e. require automatically provides the required feature, you hardly need to use a provide form explicitly. There are a couple of scenarios that you may want to use a provide form:

- To provide a feature (or features) that is/are different from the one that caused loading the file.

Suppose feature X supersedes feature Y and providing compatible APIs of Y but with different implementation. Once X.scm is loaded, you don’t want Y.scm to be loaded; so you want to tell the user that X.scm also provides the feature Y. Adding (provide "X") and (provide "Y") at the end of X.scm accomplish that. (Note: If you add a provide form, require no longer autoprovides the feature, so you need to specify (provide "X") in X.scm explicitly to provide X as well.)

Of course, this doesn’t prevent users from loading Y.scm by specifying (require "Y") before (require "X"). It should be considered just as a workaround in a production where other solutions are costly, instead of a permanent solution.
• To provide no features at all. Passing #f as feature prevents autoproviding by require without providing any feature.

This should also be a temporary solution. One possible scenario is that you are changing X.scm very frequently during development and you want (require "X") always causes loading the file. Don’t forget to remove (provide #f) when you release the file, though. Besides, for interactive reloading, consider using gauche.reload (see Section 9.29 [Reloading modules], page 508) instead.

provided? feature [Function]
Returns #t if feature is already provided.

6.22.4 Autoload

autoload file/module item . . . [Macro]
Sets up item . . . to be autoloaded. That is, when an item is referenced for the first time, file/module is loaded before the item is evaluated. This delays the loading of file/module until it is needed.

You can specify either a string file name or a symbol module name to file/module. If it is a string, the named file is loaded. If it is a symbol, the named module is loaded (using the same rule as of use), then the binding of item in the file/module is imported to the module used the autoload (See Section 4.13.3 [Defining and selecting modules], page 81, for details of use).

Item can be either a variable name (symbol), or a form (:macro symbol). If it is a variable, the named file/module is loaded when the variable is about to be evaluated. If it is the latter form, the named file/module is loaded when a form (symbol arg ...) is about to be compiled, which enables autoloading macros.

file/module must define symbol in it, or an error is signaled when file/module is autoloaded.

The following is an example of autoloading procedures.

    (autoload "foo" foo0 foo1)
    (autoload "bar" bar0 bar1)

    (define (foobar x)
        (if (list? x)
            (map bar0 x)
            (foo0)))

    (foobar '(1 2)) ; "bar" is loaded at this moment
    (foobar #f) ; "foo" is loaded at this moment

Note that if you set to autoload macro, the file/module is loaded immediately when such form that uses the macro is compiled, regardless of the piece of the code is executed or not.

6.22.5 Operations on libraries

There are several procedures you can use to check if certain libraries and/or modules are installed in the system.

In the following descriptions, pattern is either a symbol or a string. If it is a symbol, it specifies a module name (e.g. foo.bar). If it is a string, it specifies a partial pathname of the library (e.g. "foo/bar"), which will be searched under library search paths. You can also use glob-like metacharacters * and ? in pattern.
library-fold pattern proc seed :key paths strict? allow-duplicates?  

A basic iterator for library/module files. This procedure searches Scheme program files which matches pattern, under directories listed in paths (the default is the standard file load paths, *load-path*). For each matched file, it calls proc with three arguments: the matched module or library name, the full path of the program file, and the state value. Seed is used as the initial state value, and the value proc returns is used as the state value for the next call of proc. The value returned from the last proc becomes the return value of library-fold.

If pattern is a symbol and the keyword argument strict? is #t (which is the default), this procedure calls library-has-module? on the files whose name seems to match the given pattern of module name, in order to find out the file really implements the module. It can be a time consuming process if you try to match large number of modules; you can pass #f to strict? to avoid the extra check. If pattern is a string, matching is done only for file names so strict? is ignored.

By default, if there are more than one files that have the same name that matches pattern in paths, only the first one appears in paths is taken. This gives you the file you’ll get if you use require or use for that library. If you want to iterate all of matching files, pass #t to the allow-duplicates? keyword argument.

Here are some examples (the result may differ in your environment).

```
(library-fold 'srfi.1 acons '())
⇒ ((srfi.1 . "../lib/srfi/1.scm")

(library-fold "srfi.1" acons '())
⇒ ("srfi.1" . "../lib/srfi/1.scm")

;; Note the returned list is in a reverse order of
;; how acons is called.
(library-fold 'srfi.1 acons '() :allow-duplicates? #t)
⇒ ((srfi.1 . "/usr/share/gauche-0.98/0.9.13/lib/srfi/1.scm")
    (srfi.1 . "/lib/srfi/1.scm")

;; Finds available dbm implementations
(library-fold 'dbm.* acons '())
⇒ ((dbm.cdb . "/usr/share/gauche-0.98/0.9.13/lib/dbm/cdb.scm")
    (dbm.gdbm . "/lib/dbm/gdbm.scm")
    (dbm.ndbm . "/lib/dbm/ndbm.scm")
    (dbm.odbm . "/lib/dbm/odbm.scm")
```

library-map pattern proc :key paths allow-duplicates? strict?  
library-for-each pattern proc :key paths allow-duplicates? strict?  

Map and for-each version of iterator over matched libraries/modules. See library-fold above for detailed operation of matching and the meanings of keyword arguments.

Proc receives two arguments, the matched module/library name and full path of the file. Library-map returns a list of results of proc. Library-for-each discards the results.

```
(library-map 'srfi.4 list :allow-duplicates? #t)
⇒ ((srfi.4 "/lib/srfi/4.scm")
    (srfi.4 "/usr/share/gauche-0.98/0.9.13/lib/srfi/4.scm")

(library-map 'dbm.* (lambda (m p) m))
⇒ (dbm.odbm dbm.ndbm dbm.gdbm dbm.cdb)
```
library-exists? mod/path :key paths force-search? strict?  
Search a library or a module specified by mod/path, and returns a true value if it finds one. Paths and strict? keyword arguments have the same meaning as library-fold.
Unlike the iterator procedures above, this procedure first checks loaded libraries and modules in the calling process, and returns true if it finds mod/path in it, without looking into the filesystem. Passing #t to force-search? keyword arguments skips the checking of loaded libraries and modules.

library-has-module? path module  
Returns #t iff a file specified by path exists and appears to implement a module named by module. path must be an actual filename.

```
(library-has-module? "./test/foo/bar.scm" 'foo.bar)  
⇒ #t ;; if ./test/foo/bar.scm implements module foo.bar.
```

This procedure assumes a typical layout of the source code to determine if the given file implements the module, i.e., it reads the first form of the code and see if it is a define-module form that is defining the given module.

### 6.23 Sorting and merging

The interface of sorting and merging API complies SRFI-95, with the following extensions:
- You can sort not only lists, vectors and strings, but any sequence (an instance of <sequence>).
- You can use both comparison procedures and comparators (see Section 6.2.4 [Basic comparators], page 116) to specify the order.
- You can omit comparison procedure; in that case, elements are compared with default-comparator.

\[
\text{sort} \quad \text{seq :optional cmp keyfn} \\
\text{sort!} \quad \text{seq :optional cmp keyfn}
\]

[SRFI-95+] Sorts elements in a sequence seq in ascending order and returns the sorted sequence. sort! destructively reuses the original sequence.

```
You can pass an instance of any <sequence> as seq; the same type of sequence will be returned. For sort, the sequence type must have builder interface so that sort can build a new sequence of the same type (See Section 9.6.4 [Fundamental iterator creators], page 410, for the builder interface). For sort!, seq must be mutable.

The sorting order is specified by cmp. It must be either a procedure or a comparator. If it is a procedure, it must take two elements of seq, and returns #t if the first argument strictly precedes the second. If it is a comparator, it must have the comparison procedure. If omitted, default-comparator is used.

If the optional argument keyfn is given, the elements are first passed to it and the results are used for comparison. It is guaranteed that keyfn is called at most once per element.

```
(sort '(("Chopin" "Frederic")  
("Liszt" "Franz")  
("Alkan" "Charles-Valentin"))  
string<?  
car)  
⇒ (("Alkan" "Charles-Valentin")  
("Chopin" "Frederic")  
("Liszt" "Franz"))
```

In the current implementation, quicksort and heapsort algorithm is used when both cmp and keyfn is omitted, and merge sort algorithm is used otherwise. That is, the sort is stable if you
pass at least \( \texttt{cmp} \) (note that to guarantee stability, \( \texttt{cmp} \) must return \#f when given identical arguments.) SRFI-95 requires stability, but also requires \( \texttt{cmp} \) argument, so those procedures are upper-compatible to SRFI-95.

If you want to keep a sorted set of objects to which you add objects one at at time, you can also use treemaps (see Section 6.14.2 [Treemaps], page 212). If you only need to find out a few maximum or minimum elements instead of sorting all the elements, heaps can be used (see Section 12.15 [Heap], page 831).

\[
\begin{align*}
\text{sorted?} & \quad \text{seq :optional \texttt{cmp} keyfn} \\
\text{merge} & \quad \text{a b :optional \texttt{cmp} keyfn} \\
\text{merge!} & \quad \text{a b :optional \texttt{cmp} keyfn}
\end{align*}
\]

[SRFI-95+] Returns \#t iff elements in \( \text{seq} \) are in sorted order. You can pass any sequence to \( \text{seq} \). The optional argument \( \text{cmp} \) and \( \text{keyfn} \) are the same as \texttt{sort}.

In SRFI-95, \( \text{cmp} \) can’t be omitted.

\[
\begin{align*}
\text{merge} & \quad \text{a b :optional \texttt{cmp} keyfn} \\
\text{merge!} & \quad \text{a b :optional \texttt{cmp} keyfn}
\end{align*}
\]

[SRFI-95+] Arguments \( \text{a} \) and \( \text{b} \) are lists, and their elements are sorted using a compare function or a comparator \( \texttt{cmp} \). These procedures merges two list and returns a list, whose elements are sorted using \( \texttt{cmp} \). The destructive version \texttt{merge!} reuses cells in \( \text{a} \) and \( \text{b} \); the returned list is \texttt{eq?} to either \( \text{a} \) or \( \text{b} \).

In SRFI-95, \( \text{cmp} \) can’t be omitted.

The following procedures are for the backward compatibility. Their features are already covered by extended \texttt{sort} and \texttt{sort!}.

\[
\begin{align*}
\text{stable-sort} & \quad \text{seq :optional \texttt{cmp} keyfn} \\
\text{stable-sort!} & \quad \text{seq :optional \texttt{cmp} keyfn}
\end{align*}
\]

Sort a sequence \( \text{seq} \), using stable sort algorithm. Arguments \( \texttt{cmp} \) and \( \texttt{keyfn} \) are the same as \texttt{sort} and \texttt{sort!}.

In fact, \texttt{sort} and \texttt{sort!} now uses stable algorithm when \( \texttt{cmp} \) is provided, so these procedures are redundant, unless you want to omit \( \texttt{cmp} \) and yet guarantee stable sort.

\[
\begin{align*}
\text{sort-by} & \quad \text{seq \texttt{keyfn} :optional \texttt{cmp}} \\
\text{sort-by!} & \quad \text{seq \texttt{keyfn} :optional \texttt{cmp}} \\
\text{stable-sort-by} & \quad \text{seq \texttt{keyfn} :optional \texttt{cmp}} \\
\text{stable-sort-by!} & \quad \text{seq \texttt{keyfn} :optional \texttt{cmp}}
\end{align*}
\]

Variations of sort procedures that takes a key extracting function. These are redundant now, for \texttt{sort} etc. takes optional \( \texttt{keyfn} \).

### 6.24 System interface

Gauche supports most of POSIX.1 functions and other system functions popular among Unix variants as built-in procedures.

Lots of Scheme implementations provide some sort of system interface under various APIs. Some are just called by different names (e.g, \texttt{delete-file} or \texttt{remove-file} or \texttt{unlink} to delete a file), some do more abstraction introducing new Scheme objects. Instead of just picking one of such interfaces, I decided to implement Gauche’s system interface API in two layers; the lower level layer, described in this section, follows the operating system’s API as close as possible. On top of that, the higher-level APIs are provided, with considering compatibility to the existing systems.

The low level system interface has the name \texttt{sys-name} and usually correspond to the system call \texttt{name}. I tried to keep the interface similar whenever reasonable.

Gauche restarts a system call after it is interrupted by a signal by default. See Section 6.24.7 [Signal], page 306, for the details.
If you are familiar with system programming in C, see also Appendix A [C to Scheme mapping], page 1056, which shows correspondence between C standard library functions and Gauche procedures.

6.24.1 Program termination
Gauche has a few ways to terminate itself (other than returning from main). The exit procedure is a graceful way with all proper cleanups. sys-exit and sys-abort may be used in emergency where proper cleanup is impossible.

```
(exit :optional (code 0) (fmtstr #f) args ...)
```

[R7RS+] Terminates the current process with the exit code code. Code must be zero or positive exact integer. When a string is given to fmtstr, it is passed to format (see Section 6.21.8 [Output], page 275), with the rest arguments args, to produce a message to the standard error port (not the current error port; see Section 6.21.3 [Common port operations], page 260).

In fact, the exiting procedure is a bit more complicated. The precise steps of exiting is as follow.

1. The value of parameter exit-handler is checked. If it is not #f, the value is called as a procedure with three arguments: code, fmtstr, and a list of rest arguments. It is the default procedure of exit-handler that prints out the message to the standard error port. If an error occurs within exit handler, it is captured and discarded. Other exceptions are not caught.
2. The after thunks of the active dynamic winds are invoked. Any exceptions raised in after thunks are captured and discarded.
3. The clean-up handlers registered via C API Scm_AddCleanupHandler are invoked. These are usually responsible for under-the-hood cleanup jobs for each application that embeds Gauche. From the Scheme world there’s not much to care.
4. The unclosed output buffered ports are flushed.
5. The process exits with code as an exit code, via exit(3).

The exit-handler mechanism allows the application to hook its exit operation. Note that it is not for simple cleanup jobs; dynamic-wind, guard or unwind-protect are more appropriate. exit-handler is for more specific use just upon application exit. For example, GUI applications may want to post a dialog instead of printing to stderr.

For this reason, the library code shouldn’t change exit-handler; only the application knows what to do when it exits.

Another useful case is when you want to call a third-party code which calls exit inside. In that case you may swap the exit-handler for the one that raises a non-error exception while calling the third-party code. Non-error exception isn’t caught in exit, effectively interrupts the steps explained above. (Yet the after thunks of dynamic handlers are processed just like normal exception handling case.) Your application code can then capture the exception. You can use parameterize to swap exit-handler dynamically and thread-safely (see Section 6.16.1 [Parameters], page 233).

```
(guard (e [(eq? e 'exit-called) (handle-exit-as-desired)]))
  (parameterize ((exit-handler (lambda (c f a) (raise 'exit-called))))
    (call-third-party-library)))
```

Generally, calling exit while other threads are running should be avoided, since it only rewinds the dynamic handlers active in the calling threads, and other threads will be killed abruptly. If you have to do so for some reason, you may be able to use exit-handler to tell to other threads that the application is exiting. (There’s no general way, and Gauche doesn’t even have a list of all running threads; it’s application’s responsibility).
Note on design: Some languages integrate exit handling into exception handling, treating exit as a kind of exception. It is a tempting idea, so much that we’ve tried it. It didn’t work out well in Gauche; a big annoyance was that when an after thunk raised an exception during rewinding `dynamic-wind`, it shadowed the original `exit` exception.

`exit-handler :optional new-handler`  
[Function]  
When called without argument, returns the value of the current exit handler. When called with an argument, sets `new-handler` as the value of the exit handler, and returns the previous value of the exit handler. `new-handler` must be a procedure that takes three arguments, or `#f`.

The value of exit handler is thread-specific, and the default value is inherited from the value of the current exit handler of the parent thread. `exit-handler` can be used as if it’s a parameter in the `parameterize` macro (see Section 6.16.1 [Parameters], page 233).

`sys-exit code`  
[Function]  
[POSIX] Terminates the current process with the exit code `code`. `Code` must be zero or positive exact integer. This procedure calls `_exit(2)` directly. No cleanup is done. Unflushed file output is discarded.

`sys-abort`  
[Function]  
[POSIX] Calls POSIX abort(). This usually terminates the running process and dumps core. No cleanup is done.

6.24.2 Command-line arguments

The recommended way to get command-line arguments passed to a Scheme script is the argument to the `main` procedure (see Section 3.3 [Writing Scheme scripts], page 31). For the convenience, there are a few ways to access to the command-line arguments globally.

Note that a Scheme code may not always be called with a command-line argument—for example, an application-embedded Scheme scriptlet may not have the concept of command-line at all. That’s why the `main` argument is preferred, since it is an explicit interface; if `main` is called, the caller is responsible to pass in something.

That said, here are how to access the command-line arguments:

`command-line`  
[Parameter]  
[R7RS+] [SRFI-193] When called without arguments, it returns a list of command-line arguments, including the program name in the first element, as a list of strings.

When Gauche is used as an embedded language, it is application’s discretion to set up this parameter. If the application does nothing, this parameter will have an empty list. When you use this parameter in the library you have to deal with that situation.

When called with one argument, a list of string, it will become the new value of the parameter. You can use `parameterize` to switch the value of `command-line` dynamically (see Section 6.16.1 [Parameters], page 233). Note that R7RS only defines zero-argument `command-line`.

`script-file`  
[Parameter]  
[SRFI-193] While a Scheme program is run as a `script`, this parameter holds the absolute path of the Scheme source file being run. Run as a script means either it is passed as the script file to `gosh`, or the file is directly loaded from the toplevel REPL.

If this function is called from the context outside of script execution, `#f` is returned.

`*program-name*`  
[Variable]  

`*argv*`  
[Variable]  

These variables are bound to the program name and the list of command-line arguments, respectively. In Gauche scripts that are invoked by `gosh` command, `*program-name*` is usually
the name of the script, as given to gosh. When gosh is invoked interactively, *program-name* contains a zero-length string ", and *argv* is an empty list.
These variables exist in user module.
They are mainly kept for the backward compatibility. These names are compatible to STk, but other Scheme implementation uses different conventions. The command-line parameter above is preferred.
When Gauche is used as an embedded language, it’s the host application’s discretion to set up these variables. Generally, you can’t count on those variables to exist. That’s another reason you should avoid using them.

### 6.24.3 Environment inquiry

**sys-getenv name**

[Function]

[POSIX] Returns the value of the environment variable name as a string, or #f if the environment variable is not defined.

For the portable code, you may want to use SRFI-98’s get-environment-variable (see Section 11.20 [Accessing environment variables], page 727), which is also in R7RS.

Note: Most systems doesn’t guarantee thread-safety of getenv while environment is being modified; however, Gauche mutexes environment accessing/mutating APIs internally, so you don’t need to worry about the race condition as far as you use Gauche procedures.

**sys-environ**

[Function]

Returns the current environment as a list of strings. Each string is a form of NAME=VALUE, where NAME is the name of the environment variable and VALUE is its value. NAME never contains a character \=. This is useful when you want to obtain the all environment variables of the current process. Use sys-getenv if you want to query a specific environment variable.

**sys-environ->alist :optional envlist**

[Function]

A convenience procedure for sys-environ. When the list of environment strings (like what sys-environ returns) is given to envlist, this procedure splits name and value of each environment variable and returns an assoc list.

When envlist is omitted, this procedure calls sys-environ to get the current environment variables.

For the portable code, you may want to use SRFI-98’s get-environment-variables (see Section 11.20 [Accessing environment variables], page 727), which is also in R7RS.

```
(sys-environ->alist '("A=B" "C=D=E")
=> (("A" . "B") ("C" . "D=E"))
```

**sys-setenv name value :optional overwrite**

**sys-putenv name=value**

sys-setenv inserts an environment variable name with the value value. Both name and value must be a string. If the optional argument overwrite is #f (default), the environment is untouched if a variable with name already exists. If overwrite is true, the variable is overwritten.

For sys-putenv, you have to give a single string with the form of NAME=VALUE, that is, concatenating name and value with \=. If the environment variable with the same name exists, it will be overwritten.

These API reflects POSIX setenv(3) and putenv(3). However, unlike putenv(3), modifying the string passed to sys-putenv afterwards won’t affect the environment.

These procedures are only available when a feature identifier gauche.sys.setenv exists. Use cond-expand (see Section 4.12 [Feature conditional], page 75) to check their availability.
These procedures are thread-safe as far as you access and modify the environment through Gauche API.

**sys-unsetenv** *name*

Remove the environment variable with *name* (**sys-unsetenv**), or all environment variables. **sys-clearenv** is handy when you need to run subprocess, but you cannot trust the inherited environment.

These procedures are only available when a feature identifier **gauche.sys.unsetenv** exists. Use **cond-expand** (see Section 4.12 [Feature conditional], page 75) to check their availability.

```scheme
(cond-expand
[gauche.sys.unsetenv
  ... use sys-unsetenv or sys-clearenv ... ]
[else
  ... fallback code ...])
```

SRFI-98 (see Section 11.20 [Accessing environment variables], page 727) also defines a subset of above procedures to access to the environment variables. Portable programs may want to use them instead.

**gauche-version**

**gauche-architecture**

**gauche-library-directory**

**gauche-architecture-directory**

**gauche-site-library-directory**

**gauche-site-architecture-directory**

These functions returns a string that tells information about Gauche runtime itself.

**version-alist**

[SRFI-176] Returns an alist of various runtime information. The information is the same as what displayed with `gosh -V`.

**sys-available-processors**

Returns the number of available processors on the running platform. Return value is always a positive exact integer. If Gauche can’t get the information, 1 is returned.

However, if an environment variable **GAUCHE_AVAILABLE_PROCESSORS** is defined and its value can be interpreted as a positive integer, then the value is returned regardless of what the hardware/OS tells.

### 6.24.4 Filesystems

System calls that deal with filesystems. See also Section 12.34 [Filesystem utilities], page 881, which defines high-level APIs on top of the procedures described here.

### 6.24.4.1 Directories

See also Section 12.34.1 [Directory utilities], page 881, for high-level API.

**sys-readdir** *path*

*path* must be a string that denotes valid pathname of an existing directory. This function returns a list of strings of the directory entries. The returned list is not sorted. An error is signaled if *path* doesn’t exists or is not a directory.
Provides a traditional Unix glob(3) functionality; returns a list of pathnames that matches the given pattern.

This feature used to be a wrapper of system-provided glob function, hence it was named sys-glob. However, as of Gauche version 0.8.12, it was reimplemented in Scheme on top of other system calls, to overcome incompatibilities between platforms and for the opportunity to put more functionalities. So we renamed it glob. The old name sys-glob is kept for compatibility, but new programs should use glob.

The pattern argument may be a single glob pattern, or a list of glob patterns. If a list is given, pathnames that matches any one of the pattern are returned. If you’re a unix user, you already know how it works.

```
gosh> (glob "*.scm")
("ext.scm" "test.scm")
gosh> (glob "src/[ch]")
("src/ext.c" "src/ext.h")
gosh> (glob '("*.scm" "src/*.c"))
("ext.scm" "src/ext.c" "test.scm")
```

Unlike shell’s glob, if there’s no matching pathnames, () is returned.

By default, the result is sorted using built-in sort procedure (see Section 6.23 [Sorting and merging], page 291). You can pass alternative procedure to sorter argument; it should be a procedure that takes single list, and returns a sorted list. It can also be #f, in which case the result isn’t sorted at all.

In fact, globbing is a very useful tool to search hierarchical data structure in general, not limited to the filesystems. So the glob function is implemented separately from the filesystem. Using keyword arguments, you can glob from any kind of tree data structure. It is just that their default values are set to look at the filesystems.

The separator argument should be a char-set, and used to split the pattern into components. Its default is "/". It is not used to the actual pathnames to match.

The folder is a procedure that walks through the data structure. It is called with five arguments:

```
(folder proc seed parent regexp non-leaf?)
```

proc is a procedure that takes two arguments. The folder should call proc with every node in the parent whose component name matches regexp, passing around the seed value just like fold. It should return the final value returned by proc. For example, if cons is given to proc and () is given to seed, the return value of the folder is a list of nodes that matches the regexp.

The representation of a node is up to the implementation of folder. It can be a pathname, or some sort of objects, or anything. The glob procedure does not care what it is; the glob procedure merely passes the node to subsequent call to folder as parent argument, or returns a list of nodes as the result.

The parent argument is basically a node, and folder traverses its children to find the match. The exception is the initial call of folder— at the beginning glob knows nothing about each node. When glob needs to match an absolute path, it passes #t, and when glob needs to match a relative path, it passes #f, as the initial parent value.

The regexp argument is used to filter the child nodes. It should be matched against the component name of the child, not including its directory names. As a special case, it can be a symbol dir?; if that’s the case, the folder should return node itself, but it may indicate node as a directory; e.g. if node is represented as a pathname, the folder returns a pathname
with trailing directory separator. As special cases, if node is a boolean value and regexp is dir?, the folder should return the node representing root node or current node, respectively; e.g. if node is represented as a pathname, the folder may return "/" and "./" for those cases.

The non-leaf argument is a boolean flag. If it is true, the filter should omit the leaf nodes from the result (e.g. only include the directories).

Now, here’s the precise spec of glob pattern matching.

Each glob pattern is a string to match pathname-like strings. A pathname-like string is a string consists of one or more components, separated by separators. The default separator is "/"; you can change it with separator keyword argument. A component cannot contain separators, and cannot be a null string. Consecutive separators are regarded as a single separator. A pathname-like string optionally begins with, and/or ends with a separator character.

A glob pattern also consists of components and separator characters. In a component, following characters/syntax have special meanings.

* When it appears at the beginning of a component, it matches zero or more characters except a period (.). And it won’t match if the component of the input string begins with a period.
Otherwise, it matches zero or more sequence of any characters.

** If a component is just **, it matches zero or more number of components that match *. For example, src/**/*.h matches all of the following patterns.

src/*.h
src/*//*.h
src/*//*.h
src/*//*.h
...

? When it appears at the beginning of a component, it matches a character except a period (.). Otherwise, it matches any single character.

[chars] Specifies a character set. Matches any one of the set. The syntax of chars is the same as Gauche’s character set syntax (see Section 6.10 [Character sets], page 165). For the compatibility of the traditional glob, the ! character can be used to complement the character set, e.g. [!abc] is the same as [^abc].

glob-fold pattern proc seed :key separator folder sorter

This is actually a low-level construct of the glob function. Actually, glob is simply written like this:

(define (glob patterns . opts)
  (apply glob-fold patterns cons '() opts))

The meaning of pattern, separator, folder and sorter is the same as explained above.

For each pathname that matches pattern, glob-fold calls proc with the pathname and a seed value. The initial seed value is seed, and the value proc returns becomes the next seed value. The result of the last call to proc becomes the result of glob-fold. If there’s no matching pathnames, proc is never called and seed is returned.

make-glob-fs-fold :key root-path current-path

This is a utility function to generate a procedure suitable to pass the folder keyword argument of glob-fold and glob. Without arguments, this returns the same procedure which is used in glob-fold and glob by default.
The keyword arguments `root-path` and `current-path` specify the paths where `glob-fold` starts to search.

```
gosh> (glob "/tmp/\*.scm")
("/tmp/x.scm" "/tmp/y.scm")
gosh> (glob "/\*.scm"
    :folder (make-glob-fs-fold :root-path "/tmp"))
("/tmp/x.scm" "/tmp/y.scm")
gosh> (glob "/\*.scm"
    :folder (make-glob-fs-fold :current-path "/tmp"))
("/tmp/x.scm" "/tmp/y.scm")
```

See Section 6.24.4.4 [File stats], page 301, to check if a path is actually a directory.

### 6.24.4.2 Directory manipulation

- **sys-remove** `filename` [Function]  
  [POSIX] If `filename` is a file it is removed. On some systems this may also work on an empty directory, but portable scripts shouldn’t depend on it.

- **sys-rename** `old new` [Function]  
  [POSIX] Renames a file `old` to `new`. The new name can be in different directory from the old name, but both paths must be on the same device.

- **sys-tmpnam** [Function]  
  [POSIX] Creates a file name which is supposedly unique, and returns it. This is in POSIX, but its use is discouraged because of potential security risk. Use `sys-mkstemp` below if possible.

- **sys-mkstemp** `template` [Function]  
  Creates and opens a file that has unique name, and returns two values; opened port and the created filename. The file is created exclusively, avoiding race conditions. `template` is used as the prefix of the file. Unlike Unix’s `mkstemp`, you don’t need padding characters. The file is opened for writing, and its permission is set to 600.

- **sys-mkdtemp** `template` [Function]  
  Creates a directory that has unique name, and returns the name. `template` is used as the prefix of the directory. Unlike Unix’s `mkdtemp`, you don’t need padding characters. The directory’s permission is set to 700.

- **sys-link** `existing new` [Function]  
  [POSIX] Creates a hard link named `new` to the existing file `existing`.

- **sys-unlink** `pathname` [Function]  
  [POSIX] Removes `pathname`. It can’t be a directory. Returns `#t` if it is successfully removed, or `#f` if `pathname` doesn’t exist. An error is signaled otherwise.

There are similar procedures, `delete-file/remove-file` in `file.util` module, while they raises an error when the named pathname doesn’t exist (see Section 12.34.4 [File operations], page 888).

R7RS defines `delete-file`, which you may want to use in portable programs.

- **sys-symlink** `existing new` [Function]  
  Creates a symbolic link named `new` to the pathname `existing`. On systems that doesn’t support symbolic links, this function is unbound.
**sys-readlink** *path*  
[Function]  
If a file specified by *path* is a symbolic link, its content is returned. If *path* doesn’t exist or is not a symbolic link, an error is signaled. On systems that don’t support symbolic links, this function is unbound.

**sys-mkdir** *pathname* *mode*  
[Function]  
[POSIX] Makes a directory *pathname* with mode *mode*. (Note that *mode* is masked by the current umask; see **sys-umask** below). The parent directory of *pathname* must exist and be writable by the process. To create intermediate directories at once, use **make-directory** in *file.util* (Section 12.34.1 [Directory utilities], page 881).

**sys-rmdir** *pathname*  
[Function]  
[POSIX] Removes a directory *pathname*. The directory must be empty. To remove a directory with its contents, use **remove-directory** in *file.util* (Section 12.34.1 [Directory utilities], page 881).

**sys-umask** *optional* *mode*  
[Function]  
[POSIX] Sets umask setting to *mode*. Returns previous umask setting. If *mode* is omitted or #f, just returns the current umask without changing it. See man umask for more details.

### 6.24.4.3 Pathnames

See also Section 12.34.2 [Pathname utilities], page 884, for high-level APIs.

**sys-normalize-pathname** *pathname* *key absolute expand canonicalize*  
[Function]  
Converts *pathname* according to the way specified by keyword arguments. More than one keyword argument can be specified.

- **absolute**  
  If this keyword argument is given and true, and *pathname* is not an absolute pathname, it is converted to an absolute pathname by appending the current working directory in front of *pathname*.

- **expand**  
  If this keyword argument is given and true, and *pathname* begins with “~”, it is expanded as follows:
  - If *pathname* is consisted entirely by “~”, or begins with “~/”, then the character “~” is replaced for the pathname of the current user’s home directory.
  - Otherwise, characters following ‘~’ until either ‘/’ or the end of *pathname* are taken as a user name, and the user’s home directory is replaced in place of it. If there’s no such user, an error is signaled.

- **canonicalize**  
  Tries to remove pathname components “..” and “.”. The pathname interpretation is done purely in textural level, i.e. it doesn’t access filesystem to see the conversion reflects the real files. It may be a problem if there’s a symbolic links to other directory in the path.

**sys-basename** *pathname*  
[Function]  
**sys-dirname** *pathname*  
[Function]  
**sys-basename** returns a basename, that is the last component of *pathname*. **sys-dirname** returns the components of *pathname* but the last one. If *pathname* has a trailing ‘/’, it is simply ignored.

SYS-BASENAME  
SYS-DIRNAME 

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sys-basename &quot;foo/bar/bar.z&quot;)</td>
<td>&quot;bar.z&quot;</td>
</tr>
<tr>
<td>(sys-basename &quot;coo.scm&quot;)</td>
<td>&quot;coo.scm&quot;</td>
</tr>
<tr>
<td>(sys-basename &quot;x/y&quot;)</td>
<td>&quot;y&quot;</td>
</tr>
<tr>
<td>(sys-dirname &quot;foo/bar/bar.z&quot;)</td>
<td>&quot;foo/bar&quot;</td>
</tr>
</tbody>
</table>
These functions don’t check if pathname really exists.

Some boundary cases:

- `(sys-basename "") ⇒ ""
- `(sys-dirname "") ⇒ "."

- `(sys-basename "/") ⇒ ""
- `(sys-dirname "/") ⇒ "/"

Note: The above behavior is the same as Perl’s basename and dirname. On some systems, the command basename may return "/" for the argument "/", and "." for the argument ".".

**sys-realpath** *pathname*  
 `sys-realpath` returns an absolute pathname of *pathname* that does not include ".", "." or symbolic links. If *pathname* does not exist, it includes a dangling symbolic link, or the caller doesn’t have enough permission to access to the path, an error is signaled.

Note: the POSIX `realpath(3)` function is known to be unsafe, so Gauche avoids using it and implements `sys-realpath` in its own.

**sys-tmpdir**  
`sys-tmpdir` returns the default directory name suitable to put temporary files.

On Unix-like systems, the environment variable TMPDIR and TMP are first checked, then falls back to /tmp.

On Windows-native systems, it uses `GetTempPath` Windows API. It checks environment variables TMP, TEMP, and USERPROFILE in this order, and falls back to Windows system directory.

On both platforms, the returned pathname may not exist, or may not be writable by the calling process.

In general, user programs and libraries are recommended to use `temporary-directory` (see Section 12.34.1 [Directory utilities], page 881) instead; `sys-tmpdir` should be used only if you now the raw value the platform provides.

### 6.24.4.4 File stats

See also Section 12.34.3 [File attribute utilities], page 886, for high-level APIs.

**file-exists?** *path*  
`file-exists?` Returns true if *path* exists.

**file-is-regular?** *path*  
`file-is-regular?` Returns true if *path* is a regular file, or is a directory, respectively. They return false if *path* doesn’t exist at all.

**<sys-stat>**  
An object that represents struct stat, attributes of an entry in the filesystem. It has the following read-only slots.

**type**  
A symbol represents the type of the file.

- `regular` a regular file
- `directory` a directory
character  a character device
block     a block device
fifo      a fifo
 symlink   a symbolic link
 socket    a socket

If the file type is none of the above, #f is returned.

Note: Some operating systems don’t have the socket file type and returns fifo for socket files. Portable programs should check both possibilities to see if the given file is a socket.

perm      [Instance Variable of <sys-stat>]
An exact integer for permission bits of struct stat. It is the same as lower 9-bits of "mode" slot; provided for the convenience.

mode      [Instance Variable of <sys-stat>]
ino       [Instance Variable of <sys-stat>]
dev       [Instance Variable of <sys-stat>]
rdev      [Instance Variable of <sys-stat>]
nlink     [Instance Variable of <sys-stat>]
uid       [Instance Variable of <sys-stat>]
gid       [Instance Variable of <sys-stat>]
size      [Instance Variable of <sys-stat>]

An exact integer for those information of struct stat.

atime     [Instance Variable of <sys-stat>]
mtim       [Instance Variable of <sys-stat>]
ctime      [Instance Variable of <sys-stat>]

A number of seconds since Unix Epoch for those information of struct stat.

atim      [Instance Variable of <sys-stat>]
mtim       [Instance Variable of <sys-stat>]
ctim      [Instance Variable of <sys-stat>]

Timestamps represented in <time> object (see Section 6.24.9 [Time], page 316). If the system supports it, this allows up to nanosecond precision.

sys-stat path   [Function]
sys-fstat port-or-fd  [Function]

[POSIX] Returns a <sys-stat> object of path, or the underlying file of port-or-fd, which may be a port or a positive exact integer file descriptor, respectively.

If path is a symbolic link, a stat of the file the link points to is returned from sys-stat.
If port-or-fd is not associated to a file, sys-fstat returns #f.

sys-lstat path   [Function]

Like sys-stat, but it returns a stat of a symbolic link if path is a symbolic link.

goosh> (describe (sys-stat "gauche.h"))
#<sys-stat> 0x815af70> is an instance of class <sys-stat>
slots:
   type    : regular
   perm    : 420
   mode    : 33188
   ino     : 845140
   dev     : 774
   rdev    : 0
Chapter 6: Core library

nlink : 1
uid : 400
gid : 100
size : 79549
atime : 1020155914
mtime : 1020152005
ctime : 1020152005

sys-stat->mode stat
sys-stat->ino stat
sys-stat->dev stat
sys-stat->rdev stat
sys-stat->nlink stat
sys-stat->size stat
sys-stat->uid stat
sys-stat->gid stat
sys-stat->atime stat
sys-stat->mtime stat
sys-stat->ctime stat
sys-stat->file-type stat

Deprecated. Use slot-ref to access information of <sys-stat> object.

sys-access pathname amode [Function]
[POSIX] Returns a boolean value of indicating whether access of pathname is allowed in amode. This procedure signals an error if used in a suid/sgid program (see the note below). amode can be a combinations (logical or) of following predefined flags.

R_OK Checks whether pathname is readable by the current user.
W_OK Checks whether pathname is writable by the current user.
X_OK Checks whether pathname is executable (or searchable in case pathname is a directory) by the current user.
F_OK Checks whether pathname exists or not, regardless of the access permissions of pathname. (But you need to have access permissions of the directories containing pathname).

Note: Access(2) is known to be a security hole if used in suid/sgid program to check the real user’s privilege of accessing the file.

sys-chmod path mode [Function]
sys-fchmod port-or-fd mode [Function]

Change the mode of the file named path or an opened file specified by port-or-fd to mode. mode must be a small positive integer whose lower 9 bits specifies POSIX style permission.

sys-chown path owner-id group-id [Function]
Change the owner and/or group of the file named path to owner-id and group-id respectively. owner-id and group-id must be an exact integer. If either of them is -1, the corresponding ownership is not changed.

sys-utime path :optional atime mtime [Function]

Change the file’s access time and modification time to atime and mtime, respectively. The arguments can be either #f (indicating the current time), #t (do not change), a real numebr (number of seconds from Epoch), or <time> object (See Section 6.24.9 [Time], page 316).

See also touch-file (see Section 12.34.4 [File operations], page 888).
6.24.4.5 Other file operations

sys-chdir dir

[Function]
[POSIX] An interface to chdir(2). See also current-directory (see Section 12.34.1 [Directory utilities], page 881).

sys-pipe :key (buffering :line)

[Function]
[POSIX] Creates a pipe, and returns two ports. The first returned port is an input port and the second is an output port. The data put to the output port can be read from the input port.

Buffering can be :full, :line or :none, and specifies the buffering mode of the ports opened on the pipe. See Section 6.21.4 [File ports], page 264, for details of the buffering mode. The default mode is sufficient for typical cases.

(receive (in out) (sys-pipe)
 (display "abc\n" out)
 (flush out)
 (read-line in)) ⇒ "abc"

Note: the returned value is changed from version 0.3.15, in which sys-pipe returned a list of two ports.

sys-mkfifo path mode

[Function]
[POSIX] creates a fifo (named pipe) with a name path and mode mode. Mode must be a positive exact integer to represent the file mode.

sys-isatty port-or-fd

[Function]
[POSIX] port-or-fd may be a port or an integer file descriptor. Returns #t if the port is connected to the console, #f otherwise.

sys-ttyname port-or-fd

[Function]
[POSIX] port-or-fd may be a port or an integer file descriptor. Returns the name of the terminal connected to the port, or #f if the port is not connected to a terminal.

sys-truncate path length

sys-ftruncate port-or-fd length

[Function]
[POSIX] Truncates a regular file named by path or referenced by port-or-fd to a size of length bytes. If the file is larger than length bytes, the extra data is discarded. If the file is smaller than that, zero is padded.

6.24.5 Unix groups and users

Unix groups

<sys-group>

[Builtin Class]
Unix group information. Has following slots.

name
Group name.

gid
Group id.

passwd
Group password.

mem
List of user names who are in this group.


**sys-getgrgid** `gid`  
**sys-getgrnam** `name`  

[Function]  
[POSIX] Returns `<sys-group>` object from an integer group id `gid` or a group name `name`, respectively. If the specified group doesn’t exist, `#f` is returned.

**sys-gid->group-name** `gid`  
**sys-group-name->gid** `name`  

[Function]  
Convenience function to convert between group id and group name.

**Unix users**

**<sys-passwd>**  
[Builtin Class]  
Unix user information. Has following slots.

- `name`  
  User name.

- `uid`  
  User ID.

- `gid`  
  User’s primary group id.

- `passwd`  
  User’s (encrypted) password. If the system uses the shadow password file, you just get obscure string like "x".

- `gecos`  
  Gecos field.

- `dir`  
  User’s home directory.

- `shell`  
  User’s login shell.

- `class`  
  User’s class (only available on some systems).

**sys-getpwuid** `uid`  
**sys-getpwnam** `name`  

[Function]  
[POSIX] Returns `<sys-passwd>` object from an integer user id `uid` or a user name `name`, respectively. If the specified user doesn’t exist, `#f` is returned.

**sys-user-name->uid** `name`  
**sys-uid->user-name** `uid`  

[Function]  
Convenience functions to convert between user id and user name.

**Password encryption**

**sys-crypt** `key` `salt`  

[Function]  
This is the interface to `crypt(3)`. `Key` and `salt` must be a string, and an encrypted string is returned. On systems where `crypt(3)` is not available, call to this function signals an error.

This routine is only for the code that needs to check password against the system’s password database. If you are building user database on your own, you must use `crypt.bcrypt` module (see Section 12.13 [Password hashing], page 827) instead of this routine.
6.24.6 Locale

sys-setlocale  category  locale
[Function]
[POSIX] Sets/gets the locale of the category  category  to the locale  locale.  category  must be an exact integer; the following pre-defined variables are available.  locale  must be a string locale name, or #f if you’re merely querying the current locale.

Returns the locale name on success, or #f if the system couldn’t change the locale. If you pass #f to the  locale, it returns the current locale name without changing it.

LC_ALL  [Variable]
LC_COLLATE  [Variable]
LC_CTYPE  [Variable]
LC_MONETARY  [Variable]
LC_NUMERIC  [Variable]
LC_TIME  [Variable]
Predefined variables for possible  category  value of  sys-setlocale.

sys-localeconv
[Function]
[POSIX] Returns an assoc list of various information for formatting numbers in the current locale.

An example session. It may differ on your system settings.

```
(sys-localeconv)
⇒
  ((decimal_point . ".") (thousands_sep . "")
    (grouping . "") (int_curr_symbol . "")
    (currency_symbol . "") (mon_decimal_point . ")")
    (mon_thousands_sep . "") (mon_grouping . ")")
    (positive_sign . ")") (negative_sign . "")
    (int_frac_digits . 127) (frac_digits . 127)
    (p_cs_precedes . #t) (p_sep_by_space . #t)
    (n_cs_precedes . #t) (n_sep_by_space . #t)
    (p_sign_posn . 127) (n_sign_posn . 127))
```

```
(sys-setlocale LC_ALL "fr_FR")
⇒ "fr_FR"
```

```
(sys-localeconv)
⇒
  ((decimal_point . ",") (thousands_sep . "")
    (grouping . ")") (int_curr_symbol . "FRF")
    (currency_symbol . ")") (mon_decimal_point . ",")
    (mon_thousands_sep . ")") (mon_grouping . ")")
    (positive_sign . ")") (negative_sign . "")
    (int_frac_digits . 2) (frac_digits . 2)
    (p_cs_precedes . #f) (p_sep_by_space . #t)
    (n_cs_precedes . #f) (n_sep_by_space . #t)
    (p_sign_posn . 1) (n_sign_posn . 1))
```

6.24.7 Signal

Gauche can send out operating system’s signals to the other processes (including itself) and can handle the incoming signals.
In multithread environment, all threads share the signal handlers, and each thread has its own signal mask. See Section 6.24.7.5 [Signals and threads], page 312, for details.

When a system call is interrupted by a signal, and a programmer defines a handler for the signal that doesn’t transfer control to other context, the system call is restarted after the handler returns by default.

Here are some calls that are not simply restarted by signal interruption.

**close**  
This may be called through `sys-close`, `close-port`, or even implicitly when the underlying file is closed. When a signal interrupts this call, the passed fd is no longer in use. Simply restarting it can be a race if another thread just grabs the fd.

**dup2**  
This is called via `port-fd-dup!`. When `dup2` returns EINTR, the newfd is no longer in use and some other thread may grab it before restarting it.

**sleep**  
**nanosleep**  
These calls tells the remaining time when they are interrupted. We restart them with that remaining time, not the original argument, so that the total sleep time would be close to what was given originally.

On Windows native platforms, signals don’t work except some limited support of `sys-kill`.

### 6.24.7.1 Signals and signal sets

Each signal is referred by its signal number (a small integer) defined on the underlying operating system. Variables are pre-defined to the system’s signal number. System’s signal numbers may be architecture dependent, so you should use those variables rather than using literal integers.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td></td>
</tr>
<tr>
<td>SIGALRM</td>
<td></td>
</tr>
<tr>
<td>SIGCHLD</td>
<td></td>
</tr>
<tr>
<td>SIGCONT</td>
<td></td>
</tr>
<tr>
<td>SIGFPE</td>
<td></td>
</tr>
<tr>
<td>SIGHUP</td>
<td></td>
</tr>
<tr>
<td>SIGILL</td>
<td></td>
</tr>
<tr>
<td>SIGINT</td>
<td></td>
</tr>
<tr>
<td>SIGKILL</td>
<td></td>
</tr>
<tr>
<td>SIGPIPE</td>
<td></td>
</tr>
<tr>
<td>SIGQUIT</td>
<td></td>
</tr>
<tr>
<td>SIGSEGV</td>
<td></td>
</tr>
<tr>
<td>SIGSTOP</td>
<td></td>
</tr>
<tr>
<td>SIGTERM</td>
<td></td>
</tr>
<tr>
<td>SIGTSTP</td>
<td></td>
</tr>
<tr>
<td>SIGTTIN</td>
<td></td>
</tr>
<tr>
<td>SIGTTOU</td>
<td></td>
</tr>
<tr>
<td>SIGUSR1</td>
<td></td>
</tr>
<tr>
<td>SIGUSR2</td>
<td></td>
</tr>
</tbody>
</table>

These variables are bound to the signal numbers of POSIX signals.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGTRAP</td>
<td></td>
</tr>
<tr>
<td>SIGIOT</td>
<td></td>
</tr>
<tr>
<td>SIGBUS</td>
<td></td>
</tr>
<tr>
<td>SIGSTKFLT</td>
<td></td>
</tr>
<tr>
<td>SIGURG</td>
<td></td>
</tr>
<tr>
<td>SIGXCPU</td>
<td></td>
</tr>
</tbody>
</table>
SIGXFSZ  [Variable]
SIGVTALRM [Variable]
SIGPROF  [Variable]
SIGWINCH [Variable]
SIGPOLL  [Variable]
SIGIO    [Variable]
SIGPWR   [Variable]

These variables are bound to the signal numbers of system-dependent signals. Not all of them may be defined on some systems.

Besides each signal numbers, you can refer to a set of signals using a `<sys-sigset>` object. It can be used to manipulate the signal mask, and to install a signal handler to a set of signals at once.

`sys-strsignal signum`  [Function]

[POSIX] Interface to `strsignal(3)`. The argument must be an exact integer.

Returns the system message associated with the signal with the number `signum`, as a string. If no such message is available, or `signum` is not a valid signal number, this function may return `#f` or some generic message provided by the system.

`<sys-sigset>`  [Class]

A set of signals. An empty sigset can be created by

```
(make <sys-sigset>) ⇒ #<sys-sigset []>
```

`sys-sigset signal . . .`  [Function]

Creates and returns an instance of `<sys-sigset>` with members `signal . . .`. Each `signal` may be either a signal number, another `<sys-sigset>` object, or `#t` for all available signals.

```
(sys-sigset SIGHUP SIGINT) ⇒ #<sys-sigset [HUP|INT]>
```

`sys-sigset-add! sigset signal . . .`  [Function]

`sys-sigset-delete! sigset signal . . .`  [Function]

`Sigset` must be a `<sys-sigset>` object. Those procedures adds and removes the specified signals from `sigset` respectively, and returns the result. `sigset` itself is also modified.

`signal` may be either a signal number, another `<sys-sigset>` object, or `#t` for all available signals.

`sys-sigset-fill! sigset`  [Function]

`sys-sigset-empty! sigset`  [Function]

Fills `sigset` by all available signals, or empties `sigset`.

`sys-signal-name signal`  [Function]

Returns the human-readable name of the given signal number. (Note that signal numbers are system-dependent.)

```
(sys-signal-name 2) ⇒ "SIGINT"
```

6.24.7.2 Sending signals

To send a signal, you can use `sys-kill` which works like `kill(2)`.

`sys-kill pid sig`  [Function]

[POSIX] Sends a signal `sig` to the specified process(es). `Sig` must be a positive exact integer.

`pid` is an exact integer and specifies the target process(es):

- If `pid` is positive, it is the target process id.
• If $pid$ is zero, the signal is sent to every process in the process group of the current process.
• If $pid$ is less than -1, the signal is sent to every process in the process group $-pid$.

On Windows native platforms, `sys-kill` may take positive integer or a process handle (`<win:handle>` instance) as $pid$. Only `SIGKILL`, `SIGINT` and `SIGABRT` are allowed as `sig`; Gauche uses `TerminateProcess` to terminate the target process for `SIGKILL`, and sends the target process `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` for `SIGINT` and `SIGABRT`, respectively.

There's no Scheme equivalence for `raise()`, but you can use `(sys-kill (sys-getpid) sig)`.

### 6.24.7.3 Handling signals

You can register signal handling procedures in Scheme. (In multithread environment, signal handlers are shared by all threads; see Section 6.24.7.5 [Signals and threads], page 312, for details).

When a signal is delivered to the Scheme process, the VM just records it and processes it later at a 'safe point' where the state of VM is consistent. We call the signal is pending when it is registered by the VM but not processed yet.

(Note that this makes handling of some signals such as `SIGILL` useless, for the process can’t continue sensible execution after recording the signal).

If the same signal is delivered more than once before VM processes the first one, the second one and later have no effect. (This is consistent to the traditional Unix signal model.) In other words, for each VM loop a signal handler can be invoked at most once per each signal.

When too many signals of the same kind are pending, Gauche assumes something has gone wrong (e.g. infinite loop inside C-routine) and aborts the process. The default of this limit is set rather low (3), to allow unresponsive interactive script to be terminated by typing Ctrl-C three times. Note that the counter is individual for each signal; Gauche won’t abort if one `SIGHUP` and two `SIGINT`s are pending, for example. You can change this limit by `set-signal-pending-limit` described below.

When you’re using the `gosh` interpreter, the default behavior for each signal is as in the following table.

- **SIGABRT, SIGILL, SIGKILL, SIGSTOP, SIGSEGV, SIGBUS**
  Cannot be handled in Scheme. `Gosh` follows the system’s default behavior.

- **SIGCHLD, SIGTSTP, SIGTTIN, SIGTTOU, SIGWINCH**
  No signal handles are installed for these signals by `gosh`, so the process follows the system’s default behavior. A Scheme programs can install its own signal handler if necessary.

- **SIGCONT**
  If `gosh` is running with input editing mode, a signal handler is installed to restore terminal status. Otherwise, no signal handler is installed by default, and a Scheme program can install its own signal handler if necessary.

- **SIGHUP, SIGQUIT, SIGTERM**
  `Gosh` installs a signal handler for these signals that exits from the application with code 0.

- **SIGPIPE**
  `Gosh` installs a signal handler that does nothing—that is, this signal is effectively ignored by default. It is a design choice. Since Gauche delays actual handling of signals, `SIGPIPE` would be handled after the system call that tries to write to a broken pipe returns with `EPIPE`. That makes the signal a lot less useful, for we can handle the situation with error handlers for `<system-error>` with `EPIPE`. 
The default Unix behavior of **SIGPIPE** is to terminate the process. It is useful for the traditional command-line tools that are often piped together—if one of downstream commands fails, the upstream process receives **SIGPIPE** and the entire command chain is shut down without a fuss. The signal is, however, rather an annoyance for other types of output such as sockets.

Gauche does support this “exit when pipe gets stuck” convention by ports. A port can be configured as *sigpipe sensitive*; if writing to that port caused **EPIPE**, it terminates the process. By default, standard output and standard error output are configured in that way.

**SIGPWR, SIGXCPU, SIGUSR1, SIGUSR2**

On Linux platforms with thread support, these signals are used by the system and not available for Scheme. On other systems, these signals behaves the same as described below.

**other signals**

Gosh installs the default signal handler, which raises `<unhandled-signal-error>` condition (see Section 6.19.4 [Conditions], page 253). Scheme programs can override it by its own signal handler.

If you’re using Gauche embedded in some other application, it may redefine the default behavior.

Use the following procedures to get/set signal handlers from Scheme.

**set-signal-handler!** _signals handler :optional sigmask_ [Function]

Signals may be a single signal number or a `<sys-sigset>` object, and handler should be either #t, #f, #<undef>, or a procedure that takes one argument. If handler is a procedure, it will be called when the process receives one of specified signal(s), with the received signal number as an argument.

By default, the signals in _signals_ are blocked (in addition to the signal mask in effect at that time) during _handler_ is executed, so that _handler_ won’t be reentered by the same signal(s). You can provide a `<sys-sigset>` object to the _sigmask_ arg to specify the signals to be blocked explicitly. Note that the signal mask is per-thread; if more than one thread unblocks a signal, the handler may still be invoked during execution of the handler (in other thread) even if you specify _sigmask_. You have to set the threads’ signal mask properly to avoid such situation.

It is safe to do anything in _handler_, including throwing an error or invoking continuation captured elsewhere. (However, continuations captured inside _handler_ will be invalid once you return from _handler_).

If _handler_ is #t, the operating system’s default behavior is set to the specified signal(s). If _handler_ is #f, the specified signals(s) will be ignored.

If _handler_ is #<undef> (see Section 6.5 [Undefined values], page 140), it indicates Gauche to leave the current OS’s signal handler as it is. This value isn’t as much use in set-signal-handler! as in get-signal-handler: If #<undef> is passed to set-signal-handler!, it immediately returns without modifying anything. However, if you get #<undef> from get-signal-handler, you can know that the signal handler behavior hasn’t been modified by Gauche. (Note that once Gauche ever installs a signal handler, there is no way to revert back to make get-signal-handler return #<undef>).

Note that signal handler setting is shared among threads in multithread environment. The handler is called from the thread which is received the signal. See Section 6.24.7.5 [Signals and threads], page 312, for details.
get-signal-handler *signum*  [Function]
get-signal-handler-mask *signum*  [Function]
Returns the handler setting, or signal mask setting, of a signal *signum*, respectively. See set-signal-handler! for the meaning of the return value of get-signal-handler.

get-signal-handlers  [Function]
Returns an associative list of all signal handler settings. Car of each element of returned list is a `<sys-sigset>` object, and cdr of it is the handler (a procedure or a boolean value) of the signals in the set.

get-signal-pending-limit  [Function]
set-signal-pending-limit *limit*  [Function]
Gets/sets the maximum number of pending signals per each signal type. If the number of pending signals exceeds this limit, Gauche aborts the process. See the explanation at the beginning of this section for the details. *Limit* must be a nonnegative exact integer. In the current implementation the maximum number of *limit* is 255. Setting limit to zero makes the number of pending signals unlimited.

with-signal-handlers (*handler-clause ...*) *thunk*  [Macro]
A convenience macro to install signal handlers temporarily during execution of *thunk*. (Note: though this is convenient, this has certain dangerous properties described below. Use with caution.)

Each *Handler-clause* may be one of the following forms.

*(signals expr ...)*
Signals must be an expression that will yield either a signal, a list of signals, or a `<sys-sigset>` object. Installs a signal handler for signals that evaluates expr ... when one of the signals in signals is delivered.

*(signals => handler)*
This form sets the handler of signals to handler, where handler should be either #t, #f or a procedure that takes one argument.

If handler is a procedure, it will be called when the process receives one of specified signal(s), with the received signal number as an argument. If handler is #t, the operating system’s default behavior is set to the specified signal(s). If handler is #f, the specified signals(s) will be ignored.

When the control exits from *thunk*, the signal handler setting before with-signal-handlers are recovered.

CAVEAT: If you’re setting more than one signal handlers, they are installed in serial. If a signal is delivered before all the handlers are installed, the signal handler state may be left inconsistent. Also note that the handler setting is a global state; you can’t set "thread local" handler by with-signal-handlers, although the form may be misleading.

6.24.7.4 Masking and waiting signals
A Scheme program can set a signal mask, which is a set of signals to be blocked from delivery. If a signal is delivered which is completely blocked in the process, the signal becomes "pending".

The pending signal may be delivered once the signal mask is changed not to block the specified signal. (However, it depends on the operating system whether the pending signals are queued or not.)

In multithread environment, each thread has its own signal mask.
Chapter 6: Core library

sys-sigmask how mask
[Function]
Modifies the current thread’s signal mask, and returns the previous signal mask. Mask should be a <sys-sigset> object to specify the new mask, or #f if you just want to query the current mask without modifying one.

If you give <sys-sigset> object to mask, how argument should be one of the following integer constants:

SIG_SETMASK
Sets mask as the thread’s signal mask.

SIG_BLOCK
Adds signals in mask to the thread’s signal mask.

SIG_UNBLOCK
Removes signals in mask from the thread’s signal mask.

sys-sigsuspend mask
[Function]
Atomically sets thread’s signal mask to mask and suspends the calling thread. When a signal that is not blocked and has a signal handler installed is delivered, the associated handler is called, then sys-sigsuspend returns.

sys-sigwait mask
[Function]
[POSIX] Mask must be a <sys-sigset> object. If any of signals in mask is/are pending in the OS, atomically clears one of them and returns the signal number of the cleared one. If there’s no signal in mask pending, sys-sigwait blocks until any of the signals in mask arrives.

You have to block all signals in mask in all threads before calling sys-sigwait. If there’s a thread that doesn’t block the signals, the behavior of sys-sigwait is undefined.

Note: Sys-sigwait uses system’s sigwait function, whose behavior is not defined if there’s a signal handler on the signals it waits. To avoid complication, sys-sigwait resets the handlers set to the signals included in mask before calling sigwait to SIG_DFL, and restores them after sigwait returns. If another thread changes signal handlers while sys-sigwait is waiting, the behavior is undefined; you shouldn’t do that.

6.24.7.5 Signals and threads
The semantics of signals looks a bit complicated in the multithread environment. Nevertheless, it is pretty comprehensible once you remember a small number of rules. Besides, Gauche sets up the default behavior easy to use, while allowing programmers to do tricky stuff.

If you don’t want to be bothered by the details, just remember one thing, with one sidenote. By default, signals are handled by the primordial (main) thread. However, if the main thread is suspended on mutex or condition variable, the signal may not be handled at all, so be careful.

Now, if you are curious about the details, here are the rules:

• The signal handler setting is shared by all threads.
• The signal mask is thread-specific.
• If a process receives an asynchronous signal (think it as a signal delivered from other processes), one thread is chosen, out of threads which don’t block that signal.
• The signal handler is run on the chosen thread. However, if the chosen thread is waiting for acquiring a mutex lock or a condition variable, the handling of signal will be delayed until the thread is restarted. Signal delivery itself doesn’t restart the thread.

Now, these rules have several implications.

If there are more than one thread that don’t block a particular signal, you can’t know which thread receives the signal. Such a situation is much less useful in Gauche than C programs.
because of the fact that the signal handling can be delayed indefinitely if the receiver thread is waiting on mutex or condition variable. So, it is recommended to make sure, for each signal, there is only one thread that can receive it.

In Gauche, all threads created by `make-thread` (see Section 9.35.2 [Thread procedures], page 532) blocks all the signals by default (except the reserved ones). This lets all the signals to be directed to the primordial (main) thread.

Another strategy is to create a thread dedicated for handling signals. To do so, you have to block the signals in the primordial thread, then create the signal-handling thread, and within that thread you unblock all the signals. Such a thread can just loop on `sys-pause`.

```scheme
(thread-start!
  (make-thread
    (lambda ()
      (sys-sigmask SIG_SETMASK (make <sys-sigset>)) ;; empty mask
      (let loop () (sys-pause) (loop)))))
```

Complicated application may want to control per-thread signal handling precisely. You can do so, just make sure that at any moment only the designated thread unblocks the desired signal.

### 6.24.8 System inquiry

- **sys-uname** [Function] [POSIX] Returns a list of five elements, `(sysname nodename release version machine)`.
- **sys-gethostname** [Function]
  Returns the host name. If the system doesn’t have gethostname(), the second element of the list returned by `sys-uname` is used.
- **sys-getdomainname** [Function]
  Returns the domain name. If the system doesn’t have getdomainname(), "localdomain" is returned.
- **sys-getcwd** [Function] [POSIX] Returns the current working directory by a string. If the current working directory couldn’t be obtained from the system, an error is signaled. See also `sys-chdir` (see Section 6.24.4.5 [Other file operations], page 304), `current-directory` (see Section 12.34.1 [Directory utilities], page 881).
- **sys-getgid** [Function] [POSIX] Returns integer value of real and effective group id of the current process, respectively. Use `sys-gid->group-name` or `sys-getgrgid` to obtain the group’s name and other information associated to the returned group id (see Section 6.24.5 [Unix groups and users], page 304).
- **sys-setgid** [Function] `gid`
  [POSIX] Sets the effective group id of the current process.
- **sys-getuid** [Function] [POSIX] Returns integer value of real and effective user id of the current process, respectively. Use `sys-uid->user-name` or `sys-getpwuid` to obtain the user’s name and other information associated to the returned user id (see Section 6.24.5 [Unix groups and users], page 304).
- **sys-setuid** [Function] `uid`
  [POSIX] Sets the effective user id of the current process.
sys-setugid? [Function]

Returns true iff the process is running with suid/sgid-ed (that is, the program’s suid and/or sgid bit is set.)

Note: If the platform has issetugid() call, we use it. Otherwise, we remember whether the real (user|group) id and the effective (user|group) id differ or not at the initialization time. (In the latter case, we can’t detect the case that the process changes [e]uid/[e]gid before initializing Gauche; keep it in mind if libgauche is used as an embedded Scheme engine).

sys-getgroups [Function]

[POSIX] Returns a list of integer ids of supplementary groups.

sys-setgroups gids [Function]

Sets the current process’s groups to the given list of integer group ids. The caller must have the appropriate privilege.

This procedure is only available when the feature id gauche.sys.setgroups exists. Use cond-expand for the portable program:

\[
\text{(cond-expand}
\begin{align*}
\text{[gauche.sys.setgroups (sys-setgroups '(0 1))]} \\
\text{[else]}
\end{align*}
\text{)}
\]

sys-getlogin [Function]

[POSIX] Returns a string of the name of the user logged in on the controlling terminal of the current process. If the system can’t determine the information, #f is returned.

sys-getpgrp [Function]

[POSIX] Returns a process group id of the current process.

sys-getpgid pid [Function]

Returns a process group id of the process specified by pid. If pid is zero, the current process is used.

Note that getpgid() call is not in POSIX. If the system doesn’t have getpgid(), sys-getpgid still works if pid is zero (it just calls sys-getpgrp), but signals an error if pid is not zero.

sys-setpgid pid pgid [Function]

[POSIX] Sets the process group id of the process specified by pid to pgid. If pid is zero, the process ID of the current process is used. If pgid is zero, the process ID of the process specified by pid is used. (Hence sys-setpgid(0, 0) sets the process group id of the current process to the current process id).

sys-setsid [Function]

[POSIX] Creates a new session if the calling process is not a process group leader.

sys-getpid [Function]

sys-getppid [Function]

[POSIX] Returns the current process id and the parent process id, respectively.

sys-times [Function]

[POSIX]

sys-ctermid [Function]

[POSIX] Returns the name of the controlling terminal of the process. This may be just a "/dev/tty". See also sys-ttynames.
sys-getrlimit resource

sys-setrlimit resource current :optional maximum

[POSIX] Get and set resource limits respectively. Resource is an integer constant to specify
the resource of concern. The following constants are defined. (The constants marked as bsd
and/or linux indicates that they are not defined in POSIX but defined in BSD and/or Linux.
Other systems may or may not have them. Consult getrlimit manpage of your system for
the details.)

RLIMIT_AS
RLIMIT_CPU
RLIMITFSIZE
RLIMIT_MEMLOCK (bsd/linux)
RLIMIT_NICE (linux)
RLIMIT_NPROC (bsd/linux)
RLIMIT_RTPrIO (linux)
RLIMIT_SBSIZE
RLIMIT_STACK
RLIMIT_0FILE

sys-nice inc

[POSIX] Adds inc, which must be an exact integer, to the current process’s nice value. Only
the root user can specify negative inc value. Returns the new nice value.

sys-strerror errno

Errno must be an exact nonnegative integer representing a system error number. This function
returns a string describing the error.

To represent errno, the following constants are defined. Each constant is bound to an exact
integer representing the system’s error number. Note that the actual value may differ among
systems, and some of these constants may not be defined on some systems.

E2BIG EHOSTDOWN ENETDOWN ENXIO
EACCES EHOSTUNREACH ENETRESET EOPNOTSUPP
EADDRINUSE EIDRM ENETUNREACH EOVERFLOW
EADDRNOTAVAIL EINPROGRESS ENOANO EPFNOSUPPORT
EAFNOSUPPORT EINTR ENOBUFS EPIPE
EAGAIN EINVAL ENOCSI EPROTO
EALREADY EKEYEXPIRED ENOEXEC ERESTART
EAGAIN ELIBACC ENOPKG ESRCH
ECHILD ELIBBAD ENOPROTOOPT ESRMNT
ECHONG ELIBEXEC ENOSPC ESTALE
ECLOSED ELIBEXEC ENOSR ESTRPIPE
ECLOSED ELIBEXEC ENOSYS ETIMEDOUT
ECLOSED ELIBEXEC ENOTBLK ETOOMANYREFS
Chapter 6: Core library 316

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDEADLOCK</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EDESTADDRREQ</td>
<td>Destination Request</td>
</tr>
<tr>
<td>EDOM</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EDTODOT</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EDQUOT</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EEXIST</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EFBIG</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EMEDIUMTYPE</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EUCLEAN</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTEMPTY</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EWUIDBLOK</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EUNATCH</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTEMPTY</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EWOULD_BLOCK</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTUNIQ</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EXFULL</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EMSIZE</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTNAM</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EUSERS</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EDQUOT</td>
<td>Medium Type</td>
</tr>
<tr>
<td>EMULTIHOP</td>
<td>Medium Type</td>
</tr>
<tr>
<td>ENOTSOCK</td>
<td>Medium Type</td>
</tr>
</tbody>
</table>

sys-errno->symbol k

sys-symbol->errno symbol

These procedures convert between integer error number and the symbol of its unix name (e.g. EINTR).

If the given error number or name isn’t available on the running platform, those procedures return #f. See sys-strerror above for potentially available error names.

Valid error names and their actual values differ among platforms. These procedures make it easy to write portable meta-code that deal with system errors.

6.24.9 Time

Gauche has two representations of time, one is compatible to POSIX API, and the other is compatible to SRFI-18, SRFI-19 and SRFI-21. Most procedures accept both representations; if not, the representation the procedure accepts is indicated as either 'POSIX time' or 'SRFI time'.

POSIX time is represented by a real number which is a number of seconds since Unix Epoch (Jan 1, 1970, 0:00:00GMT). Procedure sys-time, which corresponds to POSIX time(2), returns this time representation.

SRFI-compatible time is represented by an object of <time> class, which keeps seconds and nanoseconds, as well as the type of the time (UTC, TAI, duration, process time, etc).

Current-time returns this representation.

POSIX time

sys-time

[Function]

[POSIX] Returns the current time in POSIX time (the time since Epoch (00:00:00 UTC, January 1, 1970), measured in seconds). It may be a non-integral number, depending on the architecture.

Note that POSIX’s definition of “seconds since the Epoch” doesn’t take leap seconds into account.

sys-gettimeofday

[Function]

Returns two values. The first value is a number of seconds, and the second value is a fraction in a number of microseconds, since 1970/1/1 0:00:00 UTC. If the system doesn’t have gettimeofday call, this function calls time(); in that case, microseconds portion is always zero.

<sys-tm>

[Builtin Class]

Represents struct tm, a calendar date. It has the following slots.

sec

[Instance Variable of <sys-tm>]

Seconds. 0-61.

min

[Instance Variable of <sys-tm>]

Minutes. 0-59.
Chapter 6: Core library

hour
   [Instance Variable of <sys-tm>]
   Hours. 0-23.

mday
   [Instance Variable of <sys-tm>]
   Day of the month, counting from 1. 1-31.

mon
   [Instance Variable of <sys-tm>]
   Month, counting from 0. 0-11.

year
   [Instance Variable of <sys-tm>]
   Years since 1900, e.g. 102 for the year 2002.

wday
   [Instance Variable of <sys-tm>]
   Day of the week. Sunday = 0 .. Saturday = 6.

yday
   [Instance Variable of <sys-tm>]
   Day of the year. January 1 = 0 .. December 31 = 364 or 365.

isdst
   [Instance Variable of <sys-tm>]
   A flag that indicates if the daylight saving time is in effect. Positive if DST is in effect, zero if not, or negative if unknown.

sys-gmtime time
   [Function]
   Converts time to <sys-tm> object, represented in GMT or local timezone, respectively. Time can be either POSIX-time or SRFI-time.

sys-localtime time
   [Function]
   [POSIX] Converts time to <sys-tm> object, represented in GMT or local timezone, respectively. Time can be either POSIX-time or SRFI-time.

sys-ctime time
   [Function]
   [POSIX] Converts time to its string representation, using POSIX ctime(). Time can be either POSIX-time or SRFI-time.

sys-difftime time1 time0
   [Function]
   [POSIX] Returns the difference of two times in the real number of seconds. Time0 and time1 can be either POSIX-time or SRFI-time.

sys-asctime tm
   [Function]
   [POSIX] Converts <sys-tm> object tm to a string representation.

sys-strftime format tm
   [Function]
   [POSIX] Converts <sys-tm> object tm to a string representation, according to a format string format. See your system’s manual entry of strftime for the supported format.
   It may return an empty string if the formatting results in an unusually lengthy string.

sys-mktime tm
   [Function]
   [POSIX] Converts <sys-tm> object tm, expressed as local time, to the POSIX-time (number of seconds since Epoch).

sys-tm->alist tm
   [Function]
   Deprecated.

SRFI time

<time>
   [Builtin Class]
   The <time> object also represents a point of time.
type
Indicates time type. **time-utc** is the default, and that represents the number of seconds since Unix Epoch. SRFI-19 (see Section 11.7 [Time data types and procedures], page 701) adds more types.

second
Second part of the time.

nanosecond
Nanosecond part of the time.

current-time
[Function]
[SRI-18][SRFI-21] Returns the **time** object representing the current time in **time-utc**. See Section 11.7 [Time data types and procedures], page 701, for it redefines **current-time** to allow optional argument to specify time type.

time? obj
[Function]
[SRI-18][SRFI-19][SRFI-21] Returns #t if obj is a time object.

seconds+ time x
[Function]
[SRFI-226] Returns a time object that is x seconds after the given time. The type of returned time object is the same as time. The argument x must be a finite real number. If it is negative, return time is before the given time.

See Section 11.7.3 [SRFI-19 Time procedures], page 702, for another way to compute with time difference by add-duration/subtract-duration.

time->seconds time
[Function]
seconds->time seconds
[Function]
[SRI-18][SRFI-21] Converts between time object and the number of seconds (POSIX-time).

Time argument of time->seconds has to be a **time** object.

6.24.10 Process management

The following procedures provide pretty raw, direct interface to the system calls. See also Section 9.27 [High-level process interface], page 488, which provides more convenient process handling on top of these primitives.

Fork and exec

sys-system command
[Function]
[POSIX] Runs command, which must be a string, in a subprocess. command is usually passed to sh, so the shell metacharacters are interpreted.

If the operating system fails to spawn a subshell, a **system-error** is thrown. Otherwise, it waits until the subshell terminates, and the integer termination status of the subshell is returned.

The termination status must be interpreted as the same way as the return value of sys-wait, described below. For example, if you want to get exit status, you have to test the result with sys-wait-exited?, then apply sys-wait-exit-status.

In general, analyzing the result of sys-system properly is cumbersome, for nonzero code may indicate subshell’s abnormal exit or signalled status or the command’s abnormal exit or signalled status. We recommend to use do-process etc. of gauche.process module if you need to handle subprocess’ exit status (see Section 9.27 [High-level process interface], page 488).

On Windows native platforms this will pass the argument to cmd.exe.
sys-fork

[Function]
[POSIX] Fork the current process. Returns 0 if you’re in the child process, and a child process’ pid if you’re in the parent process. All the opened file descriptors are shared between the parent and the child. See fork(2) of your system for details.

If the child process runs some Scheme code and exits instead of calling sys-exec, it should call sys-exit instead of exit to terminate itself. Normal exit call tries to flush the file buffers, and on some OS it messes up the parent’s file buffers.

It should be noted that sys-fork is not safe when multiple threads are running. Because fork(2) copies the process’ memory image which includes any mutex state, a mutex which is locked by another thread at the time of sys-fork remains locked in the child process, nevertheless the child process doesn’t have the thread that unlock it! (This applies to the internal mutexes as well, so even you don’t use Scheme mutex explicitly, this situation can always happen.)

If what you want is to spawn another program in a multi-threaded application, use sys-fork-and-exec explained below. If you absolutely need to run Scheme code in the child process, a typical technique is that you fork a manager process at the beginning of application, and whenever you need a new process you ask the manager process to fork one for you.

This procedure is not available on Windows native platforms.

sys-exec command args :key directory iomap sigmask

[Function]
[POSIX*] Execute command with args, a list of arguments. The current process image is replaced by command, so this function never returns.

All elements of args must be strings. The first element of args is used as argv[0], i.e. the program name.

The keyword argument directory must be a string of a directory name or #f. If it is a string, sys-exec change current working directory there before executing the program.

The iomap keyword argument, when provided, specifies how the open file descriptors are treated. It must be the following format:

((to-fd . from-port-or-fd) ...)

To-fd must be an integer, and from-port-or-fd must be an integer file descriptor or a port. Each element of the list makes the file descriptor of from-port-or-fd of the current process be mapped to the file descriptor to-fd in the executed process.

If iomap is provided, any file descriptors other than specified in the iomap list will be closed before exec(). Otherwise, all file descriptors in the current process remain open.

(sys-exec "ls" '("ls" ";")) ⇒ ;; ls is executed.

(let ((out (open-output-file "ls.out")))
  (sys-exec "ls" '("ls" ";") :iomap '((2 . 1) (1 . ,out)))
)⇒ ;; ls is executed, with its stderr redirected

;; to the current process’s stdout, and its
;; stdout redirected to the file "ls.out".

The sigmask keyword argument can be an instance of <sys-sigset> or #f (See Section 6.24.7 [Signal], page 306, for the details of signal masks). If it is an instance of <sys-sigset>, the signal mask of calling thread is replaced by it just before exec(2) is called. It is useful, for example, to run an external program from a thread where all signals are blocked (which is the default; see Section 6.24.7.5 [Signals and threads], page 312). Without setting sigmask, the execed process inherits calling thread’s signal mask and become a process that blocks all signals, which is not very convenient in most cases.
When `sys-exec` encounters an error, most of the time it raises an error condition. Once the file descriptors are permuted, however, it would be impractical to handle errors in reasonable way (you don’t even know stderr is still available!), so Gauche simply exits on the error.

On Windows native platforms, only redirections of stdin, stdout and stderr are handled. Signal mask is ignored, for Windows doesn’t have signals as the means of interprocess communication.

`sys-fork-and-exec` command args :key directory iomap sigmask

Like `sys-exec`, but executes `fork(2)` just before remapping I/O, altering signal mask and call execvp(2). Returns child’s process id. The meanings of arguments are the same as `sys-exec`.

It is strongly recommended to use this procedure instead of `sys-fork` and `sys-exec` combination when you need to spawn another program while other threads are running. No memory allocation nor lock acquisition is done between `fork(2)` and execvp(2), so it’s pretty safe in the multithreaded environment.

On Windows native platforms, this procedure returns a Windows handle object (`<win:handle>`) of the created process instead of an integer process ID. See below for Windows process handle specific API.

Like `sys-exec`, only redirections of stdin, stdout and stderr are handled on Windows native platforms.

When a true value is given to the `detached` keyword argument, the executed process is detached from the current process group and belongs to its own group. That is, it won’t be affected to the signal sent to the process group the caller process currently belongs to. It is a part of the common idioms to start a daemon process.

On Unix platforms, besides the executed process gets its own session by `setsid(2)`, it performs extra `fork(2)` to make its parent be the `init` process (pid=1). (Note: It means the running process is actually a grandchild of the calling process, although that relationship isn’t preserved. The returned pid is the running process’s one, not the intermediate process that exits immediately.)

On Windows native platforms, this flag causes the new process to be created with the `CREATE_NEW_PROCESS_GROUP` creation flag.

### Wait

`sys-wait` [Function]

[POSIX] Calls system’s `wait(2)`. The process suspends its execution until one of the child terminates. Returns two exact integer values, the first one is the child’s process id, and the second is a status code. The status code can be interpreted by the following functions.

`sys-waitpid` pid :key nohang untraced

[POSIX] This is an interface to `waitpid(3)`, an extended version of `wait`.

`pid` is an exact integer specifying which child(ren) to be waited. If it is a positive integer, it waits for that specific child. If it is zero, it waits for any member of this process group. If it is -1, it waits for any child process. If it is less than -1, it waits for any child process whose process group id is equal to the absolute value of `pid`.

If there’s no child process to wait, or a specific `pid` is given but it’s not a child process of the current process, an error (`<system-error>`, ECHILD) is signaled.

The calling process suspends until one of those child process is terminated, unless true is specified to the keyword argument `nohang`.
If true is specified to the keyword argument untraced, the status of stopped child process can be also returned.

The return values are two exact integers, the first one is the child process id, and the second is a status code. If nohang is true and no child process status is available, the first value is zero.

On Windows native platforms, this procedure may also accept a Windows process handle (<win:handle>) object as pid to wait the specific process. You can pass -1 as pid to wait for any children, but you cannot wait for a specific process group.

**sys-wait-exited? status**

[Function]

[POSIX] The argument is an exit status returned as a second value from sys-wait or sys-waitpid. sys-wait-exited? returns #t if the child process is terminated normally. sys-wait-exit-status returns the exit code the child process passed to exit(2), or the return value of main().

**sys-wait-signaled? status**

[Function]

[POSIX] The argument is an exit status returned as a second value from sys-wait or sys-waitpid. sys-wait-signaled? returns #t if the child process is terminated by an uncaught signal. sys-wait-termsig returns the signal number that terminated the child.

**sys-wait-stopped? status**

[Function]

[POSIX] The argument is an exit status returned as a second value from sys-waitpid. sys-wait-stopped? returns #t if the child process is stopped. This status can be caught only by sys-waitpid with true untraced argument. sys-wait-stopsig returns the signum number that stopped the child.

On Windows native platforms, exit code is not structured as on Unix. You cannot distinguish a process being exited voluntarily or by forced termination. Gauche uses exit code #xff09 to terminate other process with sys-kill, and the above sys-wait-\* procedures are adjusted accordingly, so that sys-wait-signaled? can likely to be used to check whether if the child process is terminated by Gauche. (See Section 6.24.7 [Signal], page 306, for the details of signal support on Windows.) Sys-wait-stopped? never returns true on Windows native platforms (yet).

**Windows specific utilities**

The following procedures are to access Windows process handle. They are only available on Windows native platforms.

**sys-win-process? obj**

[Function]

[Windows] Returns #t iff obj is a Windows process handle object.

**sys-win-process-pid handle**

[Function]

[Windows] Returns an integer PID of the process represented by a Windows process handle handle. An error is signaled if handle is not a valid Windows process handle.

Note that the API to get a pid from a process handle is only provided on or after Windows XP SP1. If you call this procedure on Windows version before that, -1 will be returned.

### 6.24.11 I/O multiplexing

The interface functions for select(2). The higher level interface is provided on top of these primitives; see Section 9.30 [Simple dispatcher], page 509.
Chapter 6: Core library

<sys-fdset> [Built-in Class]

Represents fd_set, a set of file descriptors. You can make an empty file descriptor set by make method:

(make <sys-fdset>)

sys-fdset elt . . .

Creates a new <sys-fdset> instance with file descriptors specified by elt . . .. Each elt can be an integer file descriptor, a port, or a <sys-fdset> instance. In the last case, the descriptors in the given fdset is copied to the new fdset.

sys-fdset-ref fdset port-or-fd

sys-fdset-set! fdset port-or-fd flag

Gets and sets specific file descriptor bit of fdset. port-or-fd may be a port or an integer file descriptor. If port-or-fd is a port that doesn’t have associated file descriptor, sys-fdset-ref returns #f, and sys-fdset-set! doesn’t modify fdset. flag must be a boolean value.

You can use generic setter of sys-fdset-ref as this:

(set! (sys-fdset-ref fdset port-or-fd) flag)
≡
(sys-fdset-set! fdset port-or-fd flag)

sys-fdset-copy! dest-fdset src-fdset

Copies the content of src-fdset into dest-fdset. Returns dest-fdset.

sys-fdset-clear! fdset

Empties and returns fdset.

sys-fdset->list fdset

list->sys-fdset fds

Converts an fdset to a list of integer file descriptors and vice versa. In fact, list->sys-fdset works just like (lambda (fds) (apply sys-fdset fds)), so it accepts ports and other fdsets as well as integer file descriptors.

sys-fdset-max-fd fdset

Returns the maximum file descriptor number in fdset.

sys-select readfds writefds exceptfds :optional timeout

sys-select! readfds writefds exceptfds :optional timeout

Waits for a set of file descriptors to change status. readfds, writefds, and exceptfds are <fdset> objects to represent a set of file descriptors to watch. File descriptors in readfds are watched to see if characters are ready to be read. File descriptors in writefds are watched if writing to them is ok. File descriptors in exceptfds are watched for exceptions. You can pass #f to one or more of those arguments if you don’t care about watching the condition.

timeout specifies maximum time sys-select waits for the condition change. It can be a real number, for number of microseconds, or a list of two integers, the first is the number of seconds and the second is the number of microseconds. If you pass #f, sys-select waits indefinitely.

sys-select returns four values. The first value is a number of descriptors it detected status change. It may be zero if timeout expired. The second, third and fourth values are <fdset> object that contains a set of descriptors that changed status for reading, writing, and exception, respectively. If you passed #f to one or more of readfds, writefds and exceptfds, the corresponding return value is #f.

sys-select! variant works the same as sys-select, except it modifies the passed <fdset> arguments. sys-select creates new <fdset> objects and doesn’t modify its arguments.
6.24.12 Garbage collection

The garbage collector runs implicitly whenever it is necessary, and you don’t usually need to worry about it. However, in case if you do need to worry, here are a few procedures you can use.

**gc**  
[Function]  
Trigger a full GC. It may be useful if you want to reduce interference of GC in certain parts of code by calling this immediately before that.

**gc-stat**  
[Function]  

6.24.13 Memory mapping

**<memory-region>**  
[Class]  
An object representing a mapped memory pages. It is returned from **sys-mmap**, and can be passed to **make-view-uvector** to access. It has the following read-only slots.

- **address**  
  [Instance Variable of **<memory-region>**]  
  The address of the mapping. Not useful in the Scheme world, but you may need to pass this to foreign functions.

- **size**  
  [Instance Variable of **<memory-region>**]  
  The size of the mapping, in bytes.

- **protection**  
  [Instance Variable of **<memory-region>**]  
  The bitmask represents memory protection. It is a bitwise OR of integer constants PROT_READ, PROT_WRITE, PROT_EXEC and PROT_NONE. See **sys-mmap** below for the details.

- **flags**  
  [Instance Variable of **<memory-region>**]  
  the flags passed to the **sys-mmap**. See **sys-mmap** below for the details.

**sys-mmap**  
[Function]  
[POSIX] Maps files into memory using mmap(2) (or MapViewOfFileEx on Windows). Returns a **<memory-region>** object, which can be accessed via **make-view-uvector**.

The **port** argument must be either a port or #f. If it is a port, it must have underlying file. If it is #f, the mapping becomes “anonymous” (not backed with a file); in that case, MAP_ANONYMOUS flag must also be specified in **flags**.

The **prot** argument specifies the protection bits of the memory region; it must be an integer created by bitwise-OR of the following constants:

- **PROT_EXEC**  
  The mapped pages may be executed.

- **PROT_READ**  
  The mapped pages may be read.

- **PROT_WRITE**  
  The mapped pages may be written.

- **PROT_NONE**  
  Pages may not be accessed.

The **flags** argument must be an integer created by bitwise-OR of the following constants:

- **MAP_SHARED**  
  The mapping is shared among processes. If you update the mapped pages, other processes mapping the same region can see it, too.
**MAP_PRIVATE**

The mapping is private to this process. If you update the mapped pages, it stays in the process and is not visible from outside.

**MAP_ANONYMOUS**

The pages don’t have a backing file.

The `size` argument must be a positive exact integer, and the `offset` argument must be a non-negative exact integer. They specify the region of the file to be mapped; `size` bytes starting from `offset` bytes. The `offset` must be a multiple of the page size. If `offset` is omitted, 0 is assumed. If you’re mapping non-file-backed memory (when `port` is `#f`), `offset` is ignored.

You don’t need to `unmap` the mapped pages explicitly; when the created `<memory-region>` object is garbage collected, the memory is unmapped.

**make-view-uvector mem class length :optional offset immutable?**  
This procedure creates a uniform vector that works as a “window” to the memory region `mem`, which must be a `<memory-region>` object. If the underlying memory content changes, it is visible through the uvector. If the memory region is writeable and you mutate the uvector, it is reflected to the underlying memory region.

The `class` argument must be one of the uniform vector class (e.g. `<u8vector>`), and `length` is the length of the uvector to be created. The length is the number of elements, not the number of bytes. The memory is accessed with the native endianness. `Length` can be `#f`, in that case up to the end of the memory region is used.

The `offset` argument specifies the offset from the beginning of memory region, in number of `bytes`. It must be a multiple of the uvector’s element size.

If the memory region is read-only (i.e. it is mapped without `PROT_WRITE`), the resulting uvector is immutable. Otherwise, it is mutable by default, but you can give a true value to `immutable?` argument to create an immutable uvector.

The memory region isn’t garbage collected as long as there’re uvectors that referencing the region.

### 6.24.14 Miscellaneous system calls

**sys-pause**  
[Function]  
[POSIX] Suspends the process until it receives a signal whose action is to either execute a signal-catching function or to terminate the process. This function only returns when the signal-catching function returns. The returned value is undefined.

Note that just calling `pause()` doesn’t suffice the above semantics in Scheme-level. Internally this procedure calls `sigsuspend()` with the current signal mask.

**sys-alarm seconds**  
[Function]  
[POSIX] Arranges a SIGALRM signal to be delivered after `seconds`. The previous settings of the alarm clock is canceled. Passing zero to `seconds` doesn’t schedule new alarm. Returns the number of seconds remaining until previously scheduled alarm was due to be delivered (or zero if no alarm is active).

**sys-sleep seconds :optional (no-retry #f)**  
[Function]  
[POSIX] Suspends the calling thread until the specified number of seconds elapses.

Note that libc’s `sleep(3)` could return before the specified interval if the calling thread receives a signal; in that case, `sys-sleep` automatically restarts `sleep(3)` again with remaining time interval (after invoking Scheme signal handlers if there’s any) by default. So you can count on the thread does sleep at least the specified amount of time.
If you do want \texttt{sys-sleep} to return prematurely upon receiving a signal, you can give a true value to an optional argument \texttt{no-retry}.

The reason that we retries by default is that Gauche’s GC may use signals to synchronize between threads. If GC is invoked by one thread while another thread is sleeping on \texttt{sleep(3)}, it may return prematurely. It could happen often if other threads allocate a lot, which could make \texttt{sys-sleep} unreliable.

Returns zero if it sleeps well (which is always the case if \texttt{no-retry} is false), or the number of unslept seconds if it is woke up by a signal.

To be portable across POSIX implementation, keep \textit{seconds} less than 65536.

Some systems may be using \texttt{alarm(2)} to implement \texttt{sleep(3)}, so you shouldn’t mix \texttt{sys-sleep} and \texttt{sys-alarm}.

\texttt{sys-nanosleep} \hspace{1em} \texttt{nanoseconds :optional (no-retry #f)}  
\hspace{1em} \texttt{[Function]}  
\hspace{1em} \texttt{[POSIX]} Suspends the calling thread until the specified number of nanoseconds elapses. The argument \texttt{nanoseconds} can be a \texttt{<time>} object (see Section 6.24.9 [Time], page 316), or a real number.

The system’s \texttt{nanosleep(2)} could return before the specified interval if the calling thread receives a signal; in that case, \texttt{sys-nanosleep} automatically restarts \texttt{nanosleep(2)} again with remaining time interval (after invoking Scheme signal handlers if there’s any) by default. So you can count on the thread does sleep at least the specified amount of time.

The reason that we retries by default is that Gauche’s GC may use signals to synchronize between threads. If GC is invoked by one thread while another thread is sleeping on \texttt{nanosleep(2)}, it may return prematurely. It could happen often if other threads allocate a lot, which could make \texttt{sys-nanosleep} unreliable.

Returns \#f if \texttt{nanoseconds} elapsed (which is always the case if \texttt{no-retry} is \#f), or a \texttt{<time>} object that indicates the remaining time if \texttt{sys-nanosleep} is interrupted by a signal.

; wait for 0.5 sec  
(sys-nanosleep 500000000)

; wait for 1.3 sec  
(sys-nanosleep (make <time> :second 1 :nanosecond 300000000))

Note: On Windows native platforms, this function is emulated using \texttt{Sleep}. The argument is rounded up to millisecond resolution, and it won’t be interrupted by a signal.

\texttt{sys-random}  
\hspace{1em} \texttt{[Function]}  
\hspace{1em} \texttt{sys-srandom} \hspace{1em} \texttt{seed}  
\hspace{1em} \texttt{[Function]}  
A pseudo random number generator. \texttt{sys-random} returns a random number between 0 and a positive integer \texttt{rand_max}, inclusive. This is a straightforward interface to \texttt{random(3)}. If the underlying system doesn’t have \texttt{random(3)}, \texttt{lrand48(3)} is used.

\texttt{sys-srandom} sets the seed of the random number generator. It uses either \texttt{srandom(3)} or \texttt{srand48(3)}, depending on the system.

The intention of these functions are to provide an off-the-stock handy random number generator (RNG) for applications that doesn’t sensitive to the quality and/or speed of RNG. For serious statistics analysis, use Mersenne Twister RNG in \texttt{math.mt-random} module (see Section 12.36 [Mersenne-Twister random number generator], page 893).

\texttt{RAND_MAX}  
\hspace{1em} \texttt{[Variable]}  
Bound to a positive integer that \texttt{sys-random} may return.
sys-get-osfhandle  port-or-fd  [Function]  
[Windows] This procedure is only available on Windows native platforms. Returns a Windows file handle associated to the given port or integer file descriptor. Throws an error if the given argument does not have associated file handle.

6.25 Development helper API
Gauche has some basic built-in APIs to help developers to analyze the program.

6.25.1 Debugging aid

debug-print  expr  [Macro]  
This macro prints expr in a source form, then evaluates it, then prints out the result(s), and returns them.

The output goes to the current trace port (see Section 6.21.3 [Common port operations], page 260).

The special reader syntax #?=expr is expanded into (debug-print expr). See Section 3.4 [Debugging], page 33, for the details.

debug-print-width  [Parameter]  
This parameter specifies the maximum width of information to be printed by debug-print. If the information takes more columns than the value of this parameter, it is truncated.

To show all the information, set #f to this parameter.

debug-funcall  (proc arg ...)  [Macro]  
This macro prints the value of args right before calling proc and the result(s) of the call afterwards.

The output goes to the current trace port (see Section 6.21.3 [Common port operations], page 260).

The special reader syntax #?,expr is expanded into (debug-funcall expr). See Section 3.4 [Debugging], page 33, for the details.

debug-print-conditionally  test expr  [Macro]  
debug-funcall-conditionally  test (proc arg ...)  [Macro]  
Works like debug-print and debug-funcall, but prints out the debug info only when test evaluates to true.

These are functionally equivalent to the following code:

(if test  
  (debug-print expr)  
  expr)

(if test  
  (debug-funcall (proc arg ...))  
  (proc arg ...))

The special reader syntax #??=test expr and #??,test expr are expanded into (debug-print-conditionally test expr) and (debug-funcall-conditionally test expr), respectively. See Section 3.4 [Debugging], page 33, for the details.

debug-source-info  obj  [Function]  
Retrieves source information attached to obj. The source information is returned as a list of source file name and an integer line number. If no source information is available in obj, #f is returned.
source-code closure

Returns the source code of closure, if available. Otherwise, #f is returned.

Currently, only the code that’s directly read from Scheme source is available; if the Scheme code is precompiled, the source code isn’t saved. It may be changed in future.

source-location procedure

Returns the location (a list of filename and line number) where procedure is defined, if available. Otherwise, #f is returned.

```
gosh> (use rfc.http)
gosh> (source-location http-get)
("/usr/share/gauche-0.9.0.9.5/lib/rfc/http.scm" 443)
```

If procedure is a subr or a precompiled closure, the filename may not be a complete pathname.

```
gosh> (source-location cons)
("liblist.scm" 46)
```

disasm closure

Disassemble the compiled body of closure and print it. It may not be very useful unless you’re tracking a compiler bug, or trying to tune the program to its limit.

If you’re reading the disassembler output, keep in mind that the compiled code vector may have some dead code; they are produced by the jump optimization, but the compiler doesn’t bother to eliminate them.

debug-label obj

This returns a string that is quasi-unique to an object obj. “Quasi-unique” means the label is unique to the obj— the same (eq?) objs returns the same string, and if two objs return different string they aren’t eq? to each other— until next GC occurs.

This is mostly for printing out anonymous objects that doesn’t have any other good way to distinguish each other. Note that uniqueness isn’t guaranteed across GCs, you shouldn’t use the returned value as the key to identify the objects.

6.25.2 Profiler API

These are the functions to control Gauche’s built-in profiler. See Section 3.6.1 [Using profiler], page 36, for the explanation of the profiler.

Note that the profiler isn’t guaranteed to work correctly yet in multi-threaded program, since the interaction between setitimer and threads are platform-dependent.

profiler-start

Starts the sampling profiler. If the profiler is already started, nothing is done.

profiler-stop

Stop the sampling profiler, and save the sampled data into the internal structure. If there are already saved sampled data, the newly obtained data is added to it. If the profiler isn’t running, nothing is done.

profiler-reset

Stop the profiler if it is running. Then discard the saved sampled data.

profiler-show :key sort-by max-rows

Show the saved sampled data.

The keyword argument sort-by may be one of the symbols time, count, or time-per-call, to specify how the result should be sorted. The default is time.

The keyword argument max-rows specifies the max number of rows to be shown. If it is #f, all the data is shown.
with-profile thunk

A convenience procedure. Call thunk with the sampling profiler running, and show the result to the current output port afterwards. Returns value(s) thunk yields. The profiler is reset after the result is shown.

You can’t nest this construct; the innermost with-profile will reset the profiler, invalidates any outer with-profile.
Gauche’s object system design is largely inspired by STklos, whose design has come from Tiny-CLOS. It supports multiple inheritance, multimethods, and metaobject protocol.

The type system is integrated to the object system, that is, a string is an instance of the class `<string>`, and so on.

### 7.1 Introduction to the object system

This section briefly explains the basic structure of Gauche’s object system. It is strongly influenced by CLOS (Common-Lisp Object System). If you have experience in CLOS or related systems such as TinyCLOS, STklos or Guile’s object system, you may skip to the next section.

Three concepts play the central role in CLOS-like object systems: A class, a generic function, and a method.

A class specifies a structure of object. It also defines a datatype (strictly speaking, it’s not the same thing as a datatype, but let’s skip the complicated part for now).

For example, a point in 2D space can be represented by x and y coordinates. A point class can be defined using `define-class` macro. In the shortest form, it can be defined like this:

```
(define-class <2d-point> () (x y))
```

(You can find the code of definitions in the examples of this section in `examples/oointro.scm` of Gauche’s source distribution.)

The symbol `<2d-point>` is the name of the class, and also the global variable `<2d-point>` is bound to a class object. Surrounding a class name by `<` and `>` is just a convention; you can pass any symbol to `define-class`.

The second argument of `define-class` is a list of direct superclasses, which specifies inheritance of the class. We’ll come back to it later.

The third argument of `define-class` is a list of slots. A slot is a storage space, usually in each object, where you can store a value. It is something similar to what is called a field or an instance variable in other object-oriented languages; but slots can be configured more than just a per-object storage space.

Now we defined a 2D point class, so we can create an instance of a point. You can pass a class to a generic function `make` to create an instance. (Don’t worry about what generic function is—think it as a special type of function, just for now).

```
(define a-point (make <2d-point>))
```

```
a-point ⇒ #<<2d-point> 0x8117570>
```

If you are using gosh interactively, you can use a generic function `describe` to inspect the internal of an instance. A short alias, `d`, is defined to `describe` for the convenience. (See Section 9.14 [Interactive session], page 448, for the details).

```
gosh> (d a-point)
#<<2d-point> 0x8117570> is an instance of class <2d-point>
slots:
  x : #<unbound>
  y : #<unbound>
```

In order to access or modify the value of the slot, you can use `slot-ref` and `slot-set!`, respectively. These names are taken from STklos.

```
(slot-ref a-point 'x) ;; access to the slot x of a-point
⇒ error, since slot ‘x doesn’t have a value yet
```
(slot-set! a-point 'x 10.0) ;; set 10.0 to the slot x of a-point

(slot-ref a-point 'x)
⇒ 10.0

Gauche also provides a shorter name, ref, which can also be used in SRFI-17’s generalized set! syntax:

(ref a-point 'x) ⇒ 10.0

(set! (ref a-point 'y) 20.0)

(ref a-point 'y) ⇒ 20.0

Now you can see slot values are set.

gosh> (d a-point)
#<<2d-point> 0x8117570> is an instance of class <2d-point>
slots:
x : 10.0
y : 20.0

In practice, it is usually convenient if you can specify the default value for a slot, or give values for slots when you create an instance. Such information can be specified by slot options. Let’s modify the definition of <2d-point> like this:

(define-class <2d-point> ()
  ((x :init-value 0.0 :init-keyword :x :accessor x-of)
   (y :init-value 0.0 :init-keyword :y :accessor y-of)))

Note that each slot specification is now a list, instead of just a symbol as in the previous example. The list’s car now specifies the slot name, and its cdr gives various information. The value after :init-value defines the default value of the slot. The keyword after :init-keyword defines the keyword argument which can be passed to make to initialize the slot at creation time. The name after keyword :accessor is bound to a generic function that can be used to access/modify the slot, instead of using slot-ref/slot-set!.

Let’s see some interactive session. You create an instance of the new <2d-point> class, and you can see the slots are initialized by the default values.

gosh> (define a-point (make <2d-point>))
a-point
gosh> (d a-point)
#<<2d-point> 0x8148680> is an instance of class <2d-point>
slots:
x : 0.0
y : 0.0

You create another instance, this time giving initialization values by keyword arguments.

gosh> (define b-point (make <2d-point> :x 50.0 :y -10.0))
b-point
gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
x : 50.0
y : -10.0

Accessors are less verbose than slot-ref/slot-set!, thus convenient.

gosh> (x-of a-point)
0.0
gosh> (x-of b-point)
50.0

#<undef>

Gosh> (y-of a-point)
3.33

The full list of available slot options is described in Section 7.2.1 [Defining class], page 336. At a first glance, the declarations of such slot options may look verbose. The system might have provide a static way to define init-keywords or accessor names automatically; however, CLOS-like systems prefer flexibility. Using a mechanism called metaobject protocol, you can customize how these slot options are interpreted, and you can add your own slot options as well. See Section 7.5 [Metaobject protocol], page 351, for details.

We can also have <2d-vector> class in similar fashion.

(define-class <2d-vector> ()
  ((x :init-value 0.0 :init-keyword :x :accessor x-of)
   (y :init-value 0.0 :init-keyword :y :accessor y-of)))

You can overload the method by different specializers; here you can move a point using a vector.

(gosh> (move-by! b-point 1.4 2.5)
#<undefined>

Gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
  x : 51.4
  y : -7.5

To overload the method by different specializers; here you can move a point using a vector.

(define-method move-by! ((pt <2d-point>) (delta <2d-vector>))
  (move-by! pt (x-of delta) (y-of delta)))

Specialization isn’t limited to a user-defined classes. You can also specialize a method using Gauche’s built-in type.

(define-method move-by! ((pt <2d-point>) (c <complex>))
  (move-by! pt (real-part c) (imag-part c)))

And here’s the example session:

Gosh> (define d-vector (make <2d-vector> :x -9.0 :y 7.25))
d-vector

Gosh> (move-by! b-point d-vector)
#<undefined>

Gosh> (d b-point)
#<<2d-point> 0x8155b80> is an instance of class <2d-point>
slots:
Chapter 7: Object system

\[
\begin{align*}
\text{x} & : 42.4 \\
\text{y} & : -0.25 \\
gosh> (\text{move-by! b-point 3+2i}) \\
& \text{#<undef>} \\
gosh> (\text{d b-point}) \\
& \text{#<<2d-point> 0x8155b80> is an instance of class <2d-point>} \\
& \text{slots:} \\
& \text{x} : 45.4 \\
& \text{y} : -2.25
\end{align*}
\]

You see that a method is dispatched not only by its primary receiver (<2d-point>), but also other arguments. In fact, the first argument is no more special than the rest. In CLOS-like system a method does not belong to a particular class.

So what is actually a method? Inspecting move-by! reveals that it is an instance of <generic>, a generic function. (Note that describe truncates the printed value in methods slot for the sake of readability).

\[
\begin{align*}
gosh> \text{move-by!} \\
& \text{#<generic move-by! (3)>} \\
gosh> (\text{d move-by!}) \\
& \text{#<generic move-by! (3)> is an instance of class <generic>} \\
& \text{slots:} \\
& \text{name} : \text{move-by!} \\
& \text{methods} : (#<method (move-by! <2d-point> <complex>)> #<method (move-}
\]

gosh> (ref move-by! 'methods) \\
& (#<method (move-by! <2d-point> <complex>)> \\
& #<method (move-by! <2d-point> <2d-vector>)> \\
& #<method (move-by! <2d-point> <top> <top>))>
\]

I said a generic function is a special type of function. It is recognized by Gauche as an applicable object, but when applied, it selects appropriate method(s) according to its arguments and calls the selected method(s).

What the define-method macro actually does is (1) to create a generic function of the given name if it does not exist yet, (2) to create a method object with the given specializers and the body, and (3) to add the method object to the generic function.

The accessors are also generic functions, created implicitly by the define-class macro.

\[
\begin{align*}
gosh> (\text{d x-of}) \\
& \text{#<generic x-of (2)> is an instance of class <generic>} \\
& \text{slots:} \\
& \text{name} : \text{x-of} \\
& \text{methods} : (#<method (x-of <2d-vector>)> #<method (x-of <2d-point>)>)
\end{align*}
\]

In the mainstream dynamic object-oriented languages, a class has many roles; it defines a structure and a type, creates a namespace for its slots and methods, and is responsible for method dispatch. In Gauche, namespace is managed by modules, and method dispatch is handled by generic functions.

The default printed representation of object is not very user-friendly. Gauche’s write and display function call a generic function write-object when they encounter an instance they don’t know how to print. You can define its method specialized to your class to customize how the instance is printed.

\[
\begin{align*}
& \text{(define-method write-object ((pt <2d-point>) port)} \\
& \hspace{1cm} (\text{format port } "[[~a, ~a]]" \text{ (x-of pt) (y-of pt)})
\end{align*}
\]

\[
\begin{align*}
& \text{(define-method write-object ((vec <2d-vector>) port)}
\end{align*}
\]
(format port "<<"a, "a>>" (x-of vec) (y-of vec)))

And what you’ll get is:

gosh> a-point
[[0.0, 3.33]]
gosh> d-vector
<<-9.0, 7.25>>

If you customize the printed representation to conform SRFI-10 format, and define a corresponding read-time constructor, you can make your instances to be written-out and read-back just like built-in objects. See Section 6.21.7.3 [Read-time constructor], page 273, for the details.

Several built-in functions have similar way to extend their functionality for user-defined objects. For example, if you specialize a generic function object-equal?, you can compare the instances by equal?:

(define-method object-equal? ((a <2d-point>) (b <2d-point>))
  (and (equal? (x-of a) (x-of b))
       (equal? (y-of a) (y-of b))))

(equal? (make <2d-point> :x 1 :y 2) (make <2d-point> :x 1 :y 2))
⇒ #t

(equal? (make <2d-point> :x 1 :y 2) (make <2d-point> :x 2 :y 1))
⇒ #f

(equal? (make <2d-point> :x 1 :y 2) 'a)
⇒ #f

(equal? (list (make <2d-point> :x 1 :y 2)
             (make <2d-point> :x 3 :y 4))
       (list (make <2d-point> :x 1 :y 2)
             (make <2d-point> :x 3 :y 4)))
⇒ #t

Let’s proceed to more interesting examples. Think of a class <shape>, which is an entity that can be drawn. As a base class, it keeps common attributes such as a color and line thickness in its slots.

(define-class <shape> ()
  ((color :init-value '(0 0 0) :init-keyword :color)
   (thickness :init-value 2 init-keyword :thickness)))

When an instance is created, make calls a generic function initialize, which takes care of initializing slots such as processing init-keywords and init-values. You can customize the initialization behavior by specializing the initialize method. The initialize method is called with two arguments, one is a newly created instance, and another is a list of arguments passed to make.

We define a initialize method for <shape> class, so that the created shape will be automatically recorded in a global list. Note that we don’t want to replace system’s initialize behavior completely, since we still need the init-keywords to be handled.

(define *shapes* '()) ;; global shape list

(define-method initialize ((self <shape>) initargs)
  (next-method) ;; let the system to handle slot initialization
  (push! *shapes* self)) ;; record myself to the global list


The trick is a special method, next-method. It can only be used inside a method body, and calls less specific method of the same generic function—typically, it means you call the same method of superclass. Most object-oriented languages have the concept of calling superclass’s method. Because of multiple-argument dispatching and multiple inheritance, next-method is a little bit more complicated, but the basic idea is the same.

So, what’s the superclass of <shape>? In fact, all Scheme-defined class inherits a class called <object>. And it is <object>’s initialize method which takes care of slot initialization. After calling next-method within your initialize method, you can assume all the slots are properly initialized. So it is generally the first thing in your initialize method to call next-method.

Let’s inspect the above code. When you call (make <shape> args ...), the system allocates memory for an instance of <shape>, and calls initialize generic function with the instance and args .... It is dispatched to the initialize method you just defined. In it, you call next-method, which in turn calls <object> class’s initialize method. It initializes the instance with init-values and init-keywords. After it returns, you register the new <shape> instance to the global shape list *shapes*.

The <shape> class represents just an abstract concept of shape. Now we define some concrete drawable shapes, by subclassing the <shape> class.

```
(define-class <point-shape> (<shape>)
  ((point :init-form (make <2d-point>) :init-keyword :point)))

(define-class <polyline-shape> (<shape>)
  ((points :init-value '() :init-keyword :points)
   (closed :init-value #f :init-keyword :closed)))
```

Note the second argument passed to define-class. It indicates that <point-shape> and <polyline-shape> inherit slots of <shape> class, and also instances of those subclasses can be accepted wherever an instance of <shape> class is accepted.

The <point-shape> adds one slot, point, which contains an instance of <2d-point> defined in the beginning of this section. The <polyline-shape> class stores a list of points, and a flag, which specifies whether the end point of the polyline is connected to its starting point or not.

Inheritance is a powerful mechanism that should be used with care, or it easily result a code which is untractable ("Object-oriented programming offers a sustainable way to write spaghetti code.", as Paul Graham says in his article "The Hundred-Year Language"). The rule of thumb is to make a subclass when you need a subtype. The inheritance of slots is just something that comes with, but it shouldn’t be the main reason to do subclassing. You can always "include" the substructure, as is done in <point-shape> class.

There appeared a new slot option in <point-shape> class. The :init-form slot option specifies the default value of the slot when init-keyword is not given to make method. However, unlike :init-value, with which the value is evaluated at the time the class is defined, the value with :init-form is evaluated when the system actually needs the value. So, in the <point-shape> instance, the default <2d-point> instance is only created if the <point-shape> instance is created without having :point init-keyword argument.

A shape may be drawn in different formats for different devices. For now, we just consider a PostScript output. To make the draw method polymorphic, we define a postscript output device class, <ps-device>.

```
(define-class <ps-device> () ())
```

Then we can write a draw method, specialized for both <shape> and <ps-device>.

```
(define-method draw ((self <shape>) (device <ps-device>))
  (format #t "gsave
")
  (draw-path self device)
```
In this code, the `device` argument isn't used within the method body. It is just used for method dispatching. If we eventually have different output devices, we can add a `draw` method that is specialized for such devices.

The above `draw` method does the common work, but actual drawing must be done in specialized way for each subclasses.

```scheme
(define-method draw-path ((self <point-shape>) (device <ps-device>))
  (apply format #t "newpath ~a 0 360 arc closepath\n" (point->list (ref self 'point))))

(define-method draw-path ((self <polyline-shape>) (device <ps-device>))
  (let ((pts (ref self 'points)))
    (when (>= (length pts) 2)
      (apply format #t "newpath\n")
      (apply format #t "moveto\n" (point->list (car pts)))
      (for-each (lambda (pt)
                   (apply format #t "lineto\n" (point->list pt)))
                (cdr pts))
      (when (ref self 'closed)
        (apply format #t "lineto\n" (point->list (car pts))))
      (format #t "closepath\n")))

;; utility method
(define-method point->list ((pt <2d-point>))
  (list (x-of pt) (y-of pt)))

Finally, we do a little hack. Let `draw` method work on the list of shapes, so that we can draw multiple shapes within a page in batch.

```scheme
(define-method draw ((shapes <list>) (device <ps-device>))
  (for-each (cut draw <> device) shapes)
  (format #t "showpage\n"))
```

Then we can write some simple figures ....

```scheme
(use math.const) ;; for constant pi

(define (shape-sample) ;; for constant pi
  (define (make-corners scale)
    (map (lambda (i)
           (let ((pt (make <2d-point>)))
             (move-by! pt (make-polar scale (* i 2/5 pi))
                        (move-by! pt 200 200)
                        pt))
         (iota 5)))

  (set! *shapes* '()) ;; clear the shape list
  (let* ((corners (make-corners 100)))
    ;; creates 5 corner points of pentagon
    ;; creates 5 corner points of pentagon
    (define (shape-sample) ;; for constant pi
      (define (make-corners scale)
        (map (lambda (i)
               (let ((pt (make <2d-point>)))
                 (move-by! pt (make-polar scale (* i 2/5 pi))
                            (move-by! pt 200 200)
                            pt))
              (iota 5)))

    (set! *shapes* '()) ;; clear the shape list
    (let* ((corners (make-corners 100)))
      ;; creates 5 corner points of pentagon
      (define (shape-sample) ;; for constant pi
        (define (make-corners scale)
          (map (lambda (i)
                 (let ((pt (make <2d-point>)))
                   (move-by! pt (make-polar scale (* i 2/5 pi))
                              (move-by! pt 200 200)
                              pt))
              (iota 5)))

    (set! *shapes* '()) ;; clear the shape list
  )}
Chapter 7: Object system

;;; a pentagon in green
(make <polyline-shape>
  :color '(0 1 0) :closed #t
  :points corners)

;;; a star-shape in red
(make <polyline-shape>
  :color '(1 0 0) :closed #t
  :points (list (list-ref corners 0)
               (list-ref corners 2)
               (list-ref corners 4)
               (list-ref corners 1)
               (list-ref corners 3)))

;;; put dots in each corner of the star
(for-each (cut make <point-shape> :point <>)
  (make-corners 90))

;;; draw the shapes
(draw *shapes* (make <ps-device>))

The function shape-sample writes out a PostScript code of simple drawing to the current output port. You can write it out to file by the following expression, and then view the result by PostScript viewer such as GhostScript.

(with-output-to-file "oointro.ps" shape-sample)

7.2 Class

In this section, a class in Gauche is explained in detail.

7.2.1 Defining class

To define a class, use a macro define-class.

define-class name supers (slot-spec ...) option ... [Macro]
Creates a class object according to the arguments, and globally bind it to a variable name. This macro should be used at toplevel.

Supers is a list of direct superclasses from which this class inherits. You can use multiple inheritance. All Scheme-defined classes implicitly inherits <object>. It is implicitly added to the right of supers list, so you don’t need to specify it. See Section 7.2.2 [Inheritance], page 338, for the details about inheritance.

Slot-spec is a specification of a "slot", sometimes known as a "field" or an "instance variable" (but you can specify "class variable" in slot-spec as well). The simplest form of slot-spec is just a symbol, which names the slot. Or you can give a list, whose first element is a symbol and whose rest is an interleaved list of keywords and values. The list form not only defines a name of the slot but specifies behavior of the slot. It is explained below.

Finally, option ... is an interleaved list of keywords and values, specifies how class object should be created. This macro recognizes one keyword, :metaclass, whose corresponding value is used for metaclass (class that instantiates another class). Other options are passed to the make method to create the class object. See Section 7.5.1 [Class instantiation], page 351, for the usage of metaclass.

If a slot specification is a list, it should be in the following form:

(slot-name :option1 value1 :option2 value2 ...)
Each keyword (option1 etc.) gives a slot option. By default, the following slot options are recognized. You can add more slot options by defining metaclass.

:allocation
Specifies an allocation type of this slot, which specifies how the value for this slot is stored. The following keyword values are recognized by the standard class. A programmer can define his own metaclass to extend the class to recognize other allocation types.

:instance
A slot is allocated for each instance, so that every instance can have distinct value. This realizes so-called "instance variable" behavior. If :allocation slot option is omitted, this is the default.

:class
A slot is allocated in this class object, so that every instance will share the same value for this slot. This realizes so-called "class variable" behavior. The slot value is also shared by all subclasses (unless a subclass definition shadows the slot).

:each-subclass
Similar to :class allocation, but a slot is allocated for each class; that is, it is shared by every instance of the class, but not shared by the instances of its subclasses.

:virtual
No storage is allocated for this type of slot. Accessing the slot calls procedures given in :slot-ref and :slot-set! options described below. In other words, you can make a procedural slot. If a slot’s allocation is specified as virtual, at least :slot-ref option has to be specified as well, or define-class raises an error.

:builtin
This type of allocation only appears in built-in classes, and you can’t specify it in Scheme-defined class.

:init-keyword
A keyword value given to this slot option can be used to pass an initial value to make method when an instance is created.

:init-value
Gives an initial value of the slot, if the slot is not initialized by the keyword argument at the creation time. The value is evaluated when define-class is evaluated.

:init-form
Like :init-value, but the value given is wrapped in a thunk, and evaluated each time when the value is required. If both :init-value and :init-form are given, :init-form is ignored. Actually, :init-form expr is converted to :init-thunk (lambda () expr) by define-class macro.

:initform
A synonym of :init-form. This is kept for compatibility to STk, and shouldn’t be used in the new code.

:init-thunk
Gives a thunk, which will be evaluated to obtain an initial value of the slot, if the slot is not initialized by the keyword argument at the creation time. To give a value to :init-form is equivalent to give (lambda () value) to :init-thunk.

:immutable
If not false, this slot is immutable. To be precise, you can set the slot value only once, which usually happens at instance initialization. The slot setter raises an error
unless the slot is previously unbound. If the slot is left unbound by initialize
method, it can be set later. It is regarded as a delayed initialization.

:getter Takes a symbol, and a getter method is created and bound to the generic function
of that name. The getter method takes an instance of the class and returns the
value of the slot.

:setter Takes a symbol, and a setter method is created and bound to the generic function
of that name. The setter method takes an instance of the class and a value, and
sets the value to the slot of the instance.

:accessor Takes a symbol, and create two methods; a getter method and a setter method.
A getter method is bound to the generic function of the given name, and a setter
method is added as the setter of that generic function (see Section 4.4 [Assignments],
page 52, for generic setters).

:slot-ref Specifies a value that evaluates to a procedure which takes one argument, an in-
stance. This slot option must be specified if the allocation of the slot is virtual.
Whenever a program tries to get the value of the slot, either using slot-ref or
the getter method, the specified procedure is called, and its result is returned as
the value of the slot. The procedure can return an undef value (the return value of
undefined) to indicate the slot doesn’t have a value. If the slot allocation is not
virtual this slot option is ignored.

:slot-set! Specifies a value that evaluates to a procedure which takes two arguments, an in-
stance and a value. Whenever a program tries to set the value of the slot, either
using slot-set! or the setter method, the specified procedure is called with the
value to be set. If the slot allocation is not virtual this slot option is ignored. If
this option isn’t given to a virtual slot, the slot becomes read-only.

:slot-bound? Specifies a value that evaluates to a procedure which takes one argument, an in-
stance. This slot option is only meaningful when the slot allocation is virtual.
Whenever a program tries to determine whether the slot has a value, this procedure
is called. It should return a true value if the slot has a value, or #f otherwise. If
this slot option is omitted for a virtual slot, the system calls the procedure given to
slot-ref instead, and see whether its return value is #<undef> or not.

7.2.2 Inheritance

Inheritance has two roles. First, you can extend the existing class by adding more slots. Second,
you can specialize the methods related to the existing class so that those methods will do a little
more specific task than the original methods.

Let’s define some terms. When a class <T> inherits a class <S>, we call <T> a subclass of <S>,
and <S> a superclass of <T>. This relation is transitive: <T>’s subclasses are also <S>’s subclasses,
and <S>’s superclasses are also <T>’s superclasses. Specifically, if <T> directly inherits <S>, that
is, <S> appeared in the superclass list when <T> is defined, then <S> is a direct superclass of
<T>, and <T> is a direct subclass of <S>.

When a class is defined, it and its superclasses are ordered from subclasses to superclasses,
and a list of classes is created in such order. It is called class precedence list, or CPL. Every class
has its own CPL. A CPL of a class always begins with the class itself, and ends with <top>.

You can query a class’s CPL by a procedure class-precedence-list:

gosh> (class-precedence-list <boolean>)
As you see, all classes inherits a class named `<top>`. Some built-in classes have several abstract classes in its CPL between itself and `<top>`; the above example shows `<string>` class inherits `<sequence>` and `<collection>`. That means a string can behave both as a sequence and a collection:

```scheme
(gosh> (is-a? "abc" `<string>`) #t
(gosh> (is-a? "abc" `<sequence>`) #t
(gosh> (is-a? "abc" `<collection>`) #t
```

How about inheritance of Scheme-defined classes? If there’s only single inheritance, its CPL is straightforward: you can just follow the class’s super, its super’s super, its super’s super’s super, . . ., until you reach `<top>`. See the example:

```scheme
(define-class <a> () ()
(define-class <b> (<a>) ()
(define-class <c> (<b>) ()
(define-class <d> (<c>) ()
```

Scheme-defined class always inherits `<object>`. It is automatically inserted by the system.

When multiple inheritance is involved, a story becomes a bit complicated. We have to merge multiple CPLs of the superclasses into one CPL. It is called linearization, and there are several known linearization strategies. By default, Gauche uses an algorithm called *C3 linearization*, which is consistent with the local precedence order, monotonicity, and the extended precedence graph. We don’t go into the details here; as a general rule, the order of superclasses in a class’s CPL is always consistent to the order of direct superclasses of the class, the order of CPL of each superclasses, and the order of direct superclasses of each superclass, and so on. For the precise description, see Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, P. Tucker Withington, A Monotonic Superclass Linearization for Dylan, in *Proceedings of OOPSLA 96*, October 1996.

If a class inherits superclasses in a way that its CPL can’t be constructed with satisfying consistencies, an error is reported.

Here’s a simple example of multiple inheritance.

```scheme
(define-class <grid-layout> () ()
(define-class <horizontal-grid> (<grid-layout>) ()
(define-class <vertical-grid> (<grid-layout>) ()
(define-class <hv-grid> (<horizontal-grid> <vertical-grid>) ()
```

Note that the order of direct superclasses of `<hv-grid>` ( `<horizontal-grid>` and `<vertical-grid>`) is kept.
The following is a little twisted example:

```scheme
(define-class <pane> () ())

(define-class <scrolling-mixin> () ())

(define-class <scrollable-pane> (<pane> <scrolling-mixin>) ())

(define-class <editing-mixin> () ())

(define-class <editable-pane> (<pane> <editing-mixin>) ())

(define-class <editable-scrollable-pane>
  (<scrollable-pane> <editable-pane>) ())
```

```scheme
(map class-name (class-precedence-list <editable-scrollable-pane>))
⇒ (<editable-scrollable-pane> <scrollable-pane> <editable-pane> <pane> <scrolling-mixin> <editing-mixin> <object> <top>)
```

Once the class precedence order is determined, the slots of defined class is calculated as follows: the slot definitions are collected in the direction from superclass to subclass in CPL. If a subclass has a slot definition of the same name of the one in superclass, then the slot definition of the subclass is taken and superclass’s is discarded. Suppose a class `<S>` defines slots `a`, `b`, and `c`, a class `<T>` defines slots `c`, `d`, and `e`, and a class `<U>` defines slots `b` and `e`. When `<U>`’s CPL is `<U> <T> <S> <object> <top>`), then `<U>`’s slots is calculated as the chart below; that is, `<U>` gets five slots, of which `b` and `e`’s definitions come from `<U>`’s definitions, `c` and `d`’s come from `<T>`, and `a`’s comes from `<S>`.

<table>
<thead>
<tr>
<th>CPL</th>
<th>slot definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;top&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;object&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;S&gt;</code></td>
<td><code>a</code> <code>(b)</code> <code>(c)</code></td>
</tr>
<tr>
<td><code>&lt;T&gt;</code></td>
<td><code>c</code> <code>d</code> <code>(e)</code></td>
</tr>
<tr>
<td><code>&lt;U&gt;</code></td>
<td><code>b</code> <code>e</code></td>
</tr>
<tr>
<td><code>&lt;U&gt;</code>’s slots</td>
<td><code>a</code> <code>b</code> <code>c</code> <code>d</code> <code>e</code></td>
</tr>
</tbody>
</table>

You can get a list of slot definitions of a class object using `class-slots` function.

You can get a list of slot definitions of a class object using `class-slots` function.

Note that the behavior described above is mere a default behavior. You can customize how the CPL is computed, or how slot definitions are inherited, by defining metaclass. For example, you can write a metaclass that allows you to merge slot options of the same slot names, instead of the one shadowing the other. Or you can write a metaclass that forbids a subclass shadows the superclass’s slot.

### 7.2.3 Class object

What is a class? In Gauche, a class is just an object that implements a specific feature: to instantiate an object. Because of that, you can introspect the class by just looking into the slot values. There are some procedures provided for the convenience of such introspection. Note that if those procedures return a list, it belongs to the class and you shouldn’t modify it.

```scheme
(class-name class) [Function]
```

Returns the name of `class`.
(class-name <string>) ⇒ <string>

**class-precedence-list** class  
Returns the class precedence list of class.

(class-precedence-list <string>)  
⇒ (#<class <string>>  
  #<class <sequence>>  
  #<class <collection>>  
  #<class <top>>)

**class-direct-supers** class  
Returns a list of direct superclasses of class. A direct superclass is a class from which class inherits directly.

(class-direct-supers <string>)  
⇒ (#<class <sequence>>)

**class-direct-subclasses** class  
Returns a list of direct subclasses of class. A direct subclass is a class that directly inherits class. If <T> is a direct subclass of <S>, then <S> is a direct superclass of <T>.

**class-slots** class  
Returns a list of slot definitions of class. A slot definition is a list whose car is the name of the slot and whose cdr is a keyword-value list that specifies slot options. You can further inspect a slot definition to know what characteristics the slot has. See Section 7.2.4 [Slot definition object], page 341, for the details.

The standard way to get a list of slot names of a given class is (map slot-definition-name (class-slots class)).

**class-slot-definition** class slot-name  
Returns a slot definition of a slot specified by slot-name in a class class. If class doesn’t have a named slot, #f is returned.

**class-direct-slots** class  
Returns a list of slot definitions that are directly defined in this class (i.e. not inherited from superclasses). This information is used to calculate slot inheritance during class initialization.

**class-direct-methods** class  
Returns a list of methods that has class in its specializer.

**class-slot-accessor** class slot-name  
Returns a slot accessor object of the slot specified by slot-name in class. A slot accessor object is an internal object that encapsulates the information how to access, modify, and initialize the given slot.

You don’t usually need to deal with slot accessor objects unless you are defining some special slots using metaobject protocol.

### 7.2.4 Slot definition object
A slot definition object, returned by class-slots, class-direct-slots and class-slot-definition, keeps information about a slot. Currently Gauche uses a list to represent the slot definition, as STklos and TinyCLOS do. However, it is not guaranteed that Gauche keeps such a structure in future; you should use the following dedicated accessor methods to obtain information of a slot definition object.
Chapter 7: Object system

7.2.5 Class redefinition

If the specified class name is bound to a class when define-class is used, it is regarded as redefinition of the original class.

Redefinition of a class means the following operations:

- A new class object is created based on the new definition, and bound to the variable given to define-class.
- Methods defined on the original class (i.e. methods that have the original class in their specializers) are changed so that they are defined on the new class.
- The direct-subclasses link of the direct superclasses of the original class is modified so that they will point to the new class.
- All the subclasses of the original class are redefined recursively so that they reflect the changes of the class. Each class remembers its initialization arguments, and each redefined subclass gets the same initialization arguments as the original subclass.
- The original class is marked redefined.

Note that the original class and the new class are different objects. The original class object remembers which variable in which module it is originally bound, and replaces the binding to a new class. If you keep the direct reference to the original class somewhere else, it still refers to the original class; you might want to take extra care. You can customize class redefinition behavior by defining the class-redefinition method; see Section 7.5 [Metaobject protocol], page 351, for the details.

If there are instances of the original class, such instances are automatically updated when it is about to be accessed or modified via class-of, is-a?, slot-ref, slot-set!, ref, a getter method, or a setter method.

Updating an instance means that the class of the instance is changed (from the old class to the new class). By default, the values of the slots that are common in the original class and the new class are carried over, and the slots added by the new class are initialized according to the slot specification of the new class, and the values of the slots that are removed from the original class are discarded. You can customize this behavior by writing the change-class method. See Section 7.3.3 [Changing classes], page 347, for the details.

slot-definition-name slot-def
   Returns the name of a slot given by a slot definition object slot-def.

slot-definition-options slot-def
   Returns a keyword-value list of slot options of slot-def.

slot-definition-allocation slot-def
   Returns the value of :allocation option of slot-def.

slot-definition-getter slot-def
slot-definition-setter slot-def
slot-definition-accessor slot-def
   Returns the value of :getter, :setter and :accessor slot options of slot-def, respectively.

slot-definition-option slot-def option :optional default
   Returns the value of slot option option of slot-def. If there’s no such an option, default is returned if given, or an error is signaled otherwise.
Notes on thread safety
Class redefinition process is non-local operation with full of side-effects. It is difficult to guarantee that two threads safely run class redefinition protocol simultaneously. So Gauche uses a process-wide lock to limit only one thread to enter the class redefinition protocol at a time.

If a thread tries to redefine a class while another thread is in the redefinition protocol, the thread is blocked, even if it is redefining a class different from the one that are being redefined; because redefinition affects all the subclasses, and all the methods and generic functions that are related to the class and subclasses, it is not trivial to determine two classes are completely independent or not.

If a thread tries to access an instance whose class is being redefined by another thread, also the thread is blocked until the redefinition is finished.

Note that the instance update protocol isn’t serialized. If two threads try to access an instance whose class has been redefined, both trigger the instance update protocol, which would cause an undesired race condition. It is the application’s responsibility to ensure such a case won’t happen. It is natural since the instance access isn’t serialized by the system anyway. However, an extra care is required to have mutex within an instance; just accessing the mutex in it may trigger the instance update protocol.

Notes on compatibility
Class redefinition protocols subtly differ among CLOS-like Scheme systems. Gauche’s is very similar to STklos’s, except that STklos 0.56 doesn’t replace bindings of redefined subclasses, and also it doesn’t remember initialization arguments so the redefined subclass may lose some of the information that the original subclass has. Guile’s object system swaps identities of the original class and the redefined class at the end of class redefinition protocol, so the reference to the original class object will turn to the redefined class. As far as the author knows, class redefinition is not thread-safe in both STklos 0.56 and Guile 1.6.4.

7.2.6 Class definition examples
Let’s see some examples. Suppose you are defining a graphical toolkit. A \(<\text{window}\>\) is a rectangle region on the screen, so it has width and height. It can be organized hierarchically, i.e. a window can be placed within another window; so it has a pointer to the parent window. And we specify the window’s position, \(x\), \(y\), by the coordinate relative to its parent window. Finally, we create a "root" window that covers entire screen. It also serves the default parent window. So far, what we get is something like this:

```scheme
;; The first version
(define-class <window> ()
  ;; Pointer to the parent window.
  (parent :init-keyword :parent :init-form *root-window*)
  ;; Sizes of the window
  (width :init-keyword :width :init-value 1)
  (height :init-keyword :height :init-value 1)
  ;; Position of the window relative to the parent.
  (x :init-keyword :x :init-value 0)
  (y :init-keyword :y :init-value 0))

(define *screen-width* 1280)
(define *screen-height* 1024)

(define *root-window*
```

(\)
Note the usage of :init-value and :init-form. When the <window> class is defined, we haven’t bound *root-window* yet, so we can’t use :init-value here.

```
gosh> *root-window*
#<<window> 0x80db1d0>
gosh> (define window-a (make <window> :width 100 :height 100))
window-a
```
```
gosh> (d window-a)
#<<window> 0x80db1b0> is an instance of class <window>
slots:
    parent : #<<window> 0x80db1d0>
    width  : 100
    height : 100
    x      : 0
    y      : 0
```
```
gosh> (define window-b
    (make <window> :parent window-a :width 50 :height 20 :x 10 :y 5))
window-b
```
```
gosh> (d window-b)
#<<window> 0x80db140> is an instance of class <window>
slots:
    parent : #<<window> 0x80db1b0>
    width  : 50
    height : 20
    x      : 10
    y      : 5
```

If you’re like me, you don’t want to expose a global variable such as *root-window* for users of your toolkit. One way to encapsulate it (to certain extent) is to keep the pointer to the root window in a class variable. Add the following slot option to the definition of <window>, and the slot root-window of the <window> class refers to the same storage space.

```
(define-class <window> ()
    (...)
    (root-window :allocation :class)
    (...))
```

You can use slot-ref and slot-set! on an instance of <window>, or use class-slot-ref and class-slot-set! on the <window> class itself, to get/set the value of the root-window slot.

The users of the toolkit may want to get the absolute position of the window (the coordinates in the root window) instead of the relative position. You may provide virtual slots that returns the absolute positions, like the following:

```
(define-class <window> ()
    (...)
    (root-x :allocation :virtual
        :slot-ref (lambda (o)
            (if (ref o 'parent)
                (+ (ref (ref o 'parent) 'root-x)
                   (ref o 'x))
                (ref o 'x))))
```
(:slot-set! (lambda (o v)
  (set! (ref o 'x)
    (if (ref o 'parent)
      (- v (ref (ref o 'parent) 'root-x))
      v)))
  ...
))

Whether providing such interface via methods or virtual slots is somewhat a matter of taste. Using virtual slots has an advantage of being able to hide the change of implementation, i.e. you can change to keep root-x in a real slot and make x a virtual slot later without breaking the code using <window>. (In the mainstream object-oriented languages, such kind of "hiding implementation" is usually achieved by hiding instance variables and exposing methods. In Gauche and other CLOS-like systems, slots are always visible to the users, so the situation is a bit different.

7.3 Instance

In this section, we explain how to create and use an instance.

7.3.1 Creating instance

Using class object, you can create an instance of the class by a generic function make. A specialized method for standard <class> is defined:

```
(make <class>) arg . . .
```

Creates an instance of class and returns it. Arg . . . is typically a keyword-value list to initialize the instance.

Conceptually, the default make method is defined as follows:

```
(define-method make ((class <class>) . initargs)
  (let ((obj (allocate-instance class initargs)))
    (initialize obj initargs)
    obj)
```

That is, first it allocates memory for class’s instance, then initialize it with the initialize method.

```
allocate-instance (class <class>) initargs
```

Returns a newly-allocated uninitialized instance of class.

```
initialize (obj <object>) initargs
```

The default initialize method for <object> works as follows:

- For each initializable slot of the class
  - If (the slot has :init-keyword slot option AND the keyword appears in initargs): Then the corresponding value is used to initialize the slot
  - Else if the slot has :init-value slot option: Then the value given to the slot option is used to initialize the slot
  - Else if the slot has :init-thunk slot option: Then the thunk is called, and the returned value is used to initialize the slot
  - Else: The slot is left unbound.
Among the default slot allocation classes, only instance-allocated slots are initializable and are handled by the above sequence. Class-allocated slots (e.g. its slot allocation is either :class or :each-subclass) are initialized when the class object is created, if :init-value or :init-form slot option is given. Virtual slots aren’t initialized at all.

An user-defined allocation class can be configured either initializable or not initializable; see Section 7.5 [Metaobject protocol], page 351, for the details.

If you specialize initialize method, make sure to call next-method so that the slots are properly initialized by the default sequence, before accessing any slot of the newly created instance.

Typically you specialize initialize method for your class to customize how the instance is initialized.

It is not common to specialize allocate-instance method. However, knowing that how make works, you can specialize make itself to avoid allocation of instance in some circumstances (e.g. using pre-allocated instances).

**7.3.2 Accessing instance**

**Standard accessors**

**slot-ref** *obj slot*  
Returns a value of the slot *slot* of object *obj*.

If the specified slot is not bound to any value, a generic function slot-unbound is called with three arguments, *obj*'s class, *obj*, and *slot*. The default behavior of slot-unbound is to signal an error.

If the object doesn’t have the specified slot, a generic function slot-missing is called with three arguments, *obj*'s class, *obj*, and *slot*. The default behavior of slot-missing is to signal an error.

**slot-set!** *obj slot value*  
Alters the value of the slot *slot* of object *obj* to the value *value*.

If the object doesn’t have the specified slot, a generic function slot-missing is called with four arguments, *obj*'s class, *obj*, *slot*, *value*.

**slot-bound?** *obj slot*  
Returns true if object *obj*'s slot *slot* is bound, otherwise returns false.

If the object doesn’t have the specified slot, a generic function slot-missing is called with three arguments, *obj*'s class, *obj*, *slot*.

**slot-exists?** *obj slot*  
Returns true if *obj* has the slot named *slot*.

**slot-push!** *obj slot value*  
This function implements the common idiom. It can be defined like the following code (but it may be optimized in the future versions).

```
(define (slot-push! obj slot value)
  (slot-set! obj slot (cons value (slot-ref obj slot)))
)
```

**slot-pop!** *obj slot :optional fallback*  
Reverse operation of slot-push!. If the value of *slot* of *obj* is a pair, removes its car and returns the removed item.

When the value of *slot* is not a pair, or the *slot* is unbound, *fallback* is returned if it is provided, otherwise an error is signaled.
ref (obj <object>) (slot <symbol>) [Method]
(setter ref) (obj <object>) (slot <symbol>) value [Method]

These methods just calls slot-ref and slot-set!, respectively. They are slightly less efficient than directly calling slot-ref and slot-set!, but more compact in the program code.

Fallback methods

slot-unbound [Generic Function]
slot-unbound (class <class>) obj slot [Method]

This generic function is called when an unbound slot value is retrieved. The return value of this generic function will be returned to the caller that tried to get the value.

The default method just signals an error.

slot-missing [Generic Function]
slot-missing (class <class>) obj slot :optional value [Method]

This generic function is called when a non-existent slot value is retrieved or set. The return value of this generic function will be returned to the caller that tried to get the value.

The default method just signals an error.

Special accessors

current-class-of obj [Function]

Returns a class metaobject of obj. If obj’s class has been redefined, but obj is not updated for the change, then this procedure returns the original class of obj without updating obj.

You need this procedure in rare occasions, such as within change-class method, in which you don’t want to trigger updating obj (which would cause infinite loop).

class-slot-ref class slot-name [Function]
class-slot-set! class slot-name obj [Function]
class-slot-bound? class slot-name obj [Function]

When slot’s :allocation option is either :class or :each-subclass, these procedures allow you to get/set the value of the slot without having an instance.

slot-ref-using-class (class <class>) (obj <object>) slot-name [Method]
slot-set-using-class! (class <class>) (obj <object>) slot-name value [Method]
slot-bound-using-class? (class <class>) (obj <object>) slot-name [Method]

Generic function version of slot-ref, slot-set! and slot-bound?. Class must be the class of obj.

Besides being generic, these functions are different from their procedural versions that they don’t trigger class redefinition when obj’s class has been redefined (i.e. in which case, class should be the original class of obj).

Note: Unlike CLOS, slot-ref etc. don’t call the generic function version in it, so you can’t customize the behavior of slot-ref by specializing slot-ref-using-class. So the primary purpose of those generic functions are to be used within change-class method; especially, slot-ref etc. can’t be used during obj’s being redefined, since they trigger class redefinition again (see Section 7.3.3 [Changing classes], page 347, for details).

7.3.3 Changing classes
Class change protocol

An unique feature of CLOS-family object system is that you can change classes of an existing instance. The two classes doesn’t need to be related; you can change a sewing machine into an umbrella, if you like.

change-class  [Generic Function]
change-class (obj <object>) (new-class <class>)  [Method]
Changes an object obj’s class to new-class. The default method just calls change-object-class procedure.

change-object-class obj orig-class new-class  [Function]
Changes an object obj’s class from orig-class to new-class. This isn’t a generic function—changing object’s class needs some secret magic, and this procedure encapsulates it.

The precise steps of changing class are as follow:

1. A new instance of new-class is allocated by allocate-instance.
2. For each slot of new-class:
   1. If the slot also exists in old-class, and is bound in obj, the value is retrieved from obj and set to the new instance. (The slot is carried over).
   2. Otherwise, the slot of the new instance is initialized by standard slot initialization protocol, as described in Section 7.3.1 [Creating instance], page 345.
3. Finally, the content of the new instance is transplanted to the obj—that is, obj becomes the instance of new-class without changing its identity.

Note that initialize method of new-class isn’t called on obj. If you desire, you can call it by your own change-class method.

Change-object-class returns obj.

Usually a user is not supposed to call change-object-class directly. Instead, she can define a specialized change-class. For example, if she wants to carry over the slot x of old class to the slot y of new class, she may write something like this:

(define-method change-class ((obj <old-class>) <new-class>)
(let ((old-val (slot-ref obj 'x)))
 (next-method) ;; calls default change-class
 (slot-set! obj 'y old-val) ;; here, obj’s class is already <new-class>. obj))

Customizing instance update

Updating an instance for a redefined class is also handled as class change. When an object is accessed via normal slot accessor/modifier, its class is checked whether it has been redefined. And if it has indeed been redefined, change-class is called with the redefined class as new-class; that is, updating an instance is regarded as changing object’s class from the original one to the redefined one.

By specializing change-class, you can customize the way an instance is updated for a redefined class. However, you need a special care to write change-class for class redefinition.

First, the redefinition changes global binding of the class object. So you need to keep the reference to the old class before redefining the class, and use the old class to specialize change-class method:

;;; save old <myclass>
(define <old-myclass> <myclass>)
Chapter 7: Object system

```scheme
;; redefine <myclass>
(define-class <myclass> ()
  ...)

;; define customized change-class method
(define-method change-class ((obj <old-myclass>) <myclass>)
  ...
  (next-method)
  ...)

Next, note that the above change-class method may be triggered implicitly when you access to obj via slot-ref, slot-set!, class-of, etc. If you use such procedures like slot-ref on obj again within change-class, it would trigger the instance update protocol recursively, which would cause an infinite loop. You can only use the methods that doesn’t trigger instance update, that is, slot-ref-using-class, slot-set-using-class!, slot-bound-using-class? and current-class-of.

If you want to carry over a slot whose value is calculated procedurally, such as a virtual slot, then slot-ref etc. might be called implicitly on obj during calculating the slot value. Actually change-object-class has a special protection to detect such a recursion. If that happens, change-object-class gives up to retrieve the slot value and just initializes the slot of the new instance as if the old slot were unbound.

Customizing instance update is highly tricky business, although very powerful. You can find some nontrivial cases in the test program of Gauche source code; take a look at test/object.scm.

7.4 Generic function and method

Defining methods

define-generic name :key class
  [Macro]
  Creates a generic function and bind it to name.
  You don’t usually need to use this, since the define-method macro implicitly creates a generic function if it doesn’t exist yet.
  You can pass a subclass of <generic> to the class keyword argument so that the created generic function will be the instance of the passed class, instead of the default <generic> class. It is useful when you defined a subclass of <generic> to customize generic function application behavior.

define-method name [qualifier ...] specs body
  [Macro]
  Defines a method whose name is name. If there’s already a generic function object globally bound to name, the created method is added to the generic function. If name is unbound, or bound to an object except a generic function, then a new generic function is created, bound to name, then a new method is added to it.
  The name can be followed by optional qualifiers, each of which is a keyword. Currently, only the following qualifier is valid.

:locked
  Declares that you won’t redefine the method with the same specifiers. Attempt to redefine it will raise an error. (You can still define methods with different specifiers.)
  Most methods concerning basic operations on built-in objects are locked, for redefining them would case Gauche’s infrastructure unstable. It also allows Gauche to perform certain optimizations.
Specs specifies the arguments and their types for this method. It’s like the argument list of lambda form, except you can specify the type of each argument.

\[
specs : ( \text{arg} \ldots ) \\
| ( \text{arg} \ldots . \text{symbol} ) \\
| ( \text{arg} \ldots \text{extended-spec} \ldots ) \\
| \text{symbol}
\]

\[
arg : ( \text{symbol class} ) \\
| \text{symbol}
\]

Class specifies the class that the argument has to belong to. If arg is just a symbol, it is equivalent to (arg <top>). You can’t specify the type for the “rest” argument, for it is always bound to a list.

You can use extended argument specifications such as :optional, :key and :rest as well. (See Section 4.3 [Making procedures], page 47, for the explanation of extended argument specifications). Those extended arguments are treated as if a single “rest” argument in terms of dispatching; they aren’t used for method dispatch, and you can’t specify classes for these optional and keyword arguments.

The list of classes of the argument list is called method specializer list, based on which the generic function will select appropriate methods(s). Here are some examples of specs and the corresponding specializer list (note that the rest argument isn’t considered as a part of specializer list; we know it’s always a list.) The optional item indicates whether the method takes rest arguments or not.

\[
specs: ((\text{self <myclass>}) (\text{index <integer>}) \text{value}) \\
specializers: (<\text{myclass} > <\text{integer} > <\text{top}>) \\
optional: \#f
\]

\[
specs: (\text{obj} (\text{attr <string>})) \\
specializers: (<\text{top} > <\text{string}>) \\
optional: \#f
\]

\[
specs: ((\text{self <myclass>}) \text{obj} . \text{options}) \\
specializers: (<\text{myclass} > <\text{top}>) \\
optional: \#t
\]

\[
specs: ((\text{self <myclass>}) \text{obj} :\text{optional} (a 0) (b 1) :\text{key} (c 2)) \\
specializers: (<\text{myclass} > <\text{top}>) \\
optional: \#t
\]

\[
specs: \text{args} \\
specializers: () \\
optional: \#t
\]

If you define a method on name whose specializer list, and whether it takes rest arguments, match with one in the generic function’s methods, then the existing method is replaced by the newly defined one, unless the original method is locked.

Note: If you’re running Gauche with keyword-symbol integrated mode (see Section 6.8.1 [Keyword and symbol integration], page 158), there’s an ambiguity if you specify a keyword as the sole specs (to receive entire arguments in a single variable). Gauche parses keywords following name as qualifiers, so avoid using a keyword as such a variable.
Applying generic function

When a generic function is applied, first it selects methods whose specializer list matches the given arguments. For example, suppose a generic function `foo` has three methods, whose specializer lists are `(string top)`, `(string string)`, and `(top top)`, respectively. When `foo` is applied like `(foo "abc" 3)`, the first and the third method will be selected.

Then the selected methods are sorted from the most `specific` method to the least specific method. It is calculated as follows:

- Suppose we have a method `a` that has specializers (A1 A2 ...), and a method `b` that has (B1 B2 ...).
- Find the minimum `n` where the classes `An` and `Bn` differ. Then the class of `n`-th argument is taken, and its class precedence list is checked. If `An` comes before `Bn` in the CPL, then method `a` is more specific than `b`. Otherwise, `b` is more specific than `a`.
- If all the specializers of `a` and `b` are the same, except that one has an improper tail (`"rest"` argument) and another doesn’t, then the method that doesn’t have an improper tail is more specific than the one that has.

Once methods are sorted, the body of the first method is called with the actual argument. Within the method body, a special local variable `next-method` is bound implicitly.

```
next-method                [Next method]
next-method args ...      [Next method]
```

This variable is bound within a method body to a special object that encapsulates the next method in the sorted method list.

Calling without arguments invokes the next method with the same arguments as this method is called with. Passing `args ...` explicitly invokes the next method with the passed arguments.

If `next-method` is called in the least specific method, i.e. there’s no "next method", an error is signaled.

7.5 Metaobject protocol

In CLOS-like object systems, the object system is built on top of itself—that is, things such as the structure of the class, how a class is created, how an instance is created and initialized, and how a method is dispatched and called, are all defined in terms of the object system. For example, a class is just an instance of the class `<class>` that defines a generic structure and behavior of standard classes. If you subclass `<class>`, then you can create your own set of classes that behaves differently than the default behavior; in effect, you are creating your own object system.

`Metaobject protocols` are the definitions of APIs concerning about how the object systems are built—building-block classes, and the names and orders of generic functions to be called during operations of the object system. Subclassing these classes and specializing these methods are the means of customizing object system behaviors.

7.5.1 Class instantiation

Every class is an instance of a group of special classes. A class that can be a class of another class is called `metaclass`. In Gauche, only the `<class>` class or its subclasses can be a metaclass.

Expansion of `define-class`

The `define-class` macro is basically a wrapper of the code that creates an instance of `<class>` (or specified metaclass) and bind it to the given name. Suppose you have the following `define-class` form.

```
(define-class name (supers)
```
slot-specs
options ...

It is expanded into a form like this (you can see the exact form by looking at the definition of define-class macro in src/libobj.scm of the source code tree.

(define name
  (let ((tmp1 (make metaclass
:  :name 'name :supers (list supers)
:  :slots (map process-slot-definitions
           slot-specs)
:  :defined-modules (list (current-module))
           options ...)))
... check class redefinition ...
... registering accessor methods ...
tmp1))

The created class’s class, i.e. metaclass, is determined by the following rules.

1. If :metaclass option is given to the define-class macro, its value is used. The value must be the <class> class or its descendants.

2. Otherwise, the metaclasses of the classes in the class precedence list is examined.
   • If all the metaclasses are <class>, then the created class’s metaclass is also <class>.
   • If all the metaclasses are either <class> or another metaclass A, then the created class’ metaclass is A.
   • If the set of metaclasses contains more than one metaclass (A, B, C . . .) other than <class>, then the created class’ metaclass is a metaclass that inherits all of those metaclasses A, B, C . . . .

The class’s name, superclasses, and slot definitions are passed as the initialization arguments to the make generic function, with other arguments passed to define-class. The initialization argument defined-modules is passed to remember which module the class is defined, for the redefinition of this class.

The slot specifications slot-specs are processed by internal method process-slot-definitions (which can’t be directly called) to be turned into slot definitions. Specifically, an :init-form slot option is turned into an :init-thunk option, and :getter, :setter and :accessor slot options are quoted.

After the class (an instance of metaclass) is created, the global binding of name is checked. If it is bound to a class, then the class redefinition protocol is invoked (see Section 7.2.5 [Class redefinition], page 342).

Then, the methods given to :getter, :setter and :accessor slot options in slot-spec are collected and registered to the corresponding generic functions.

Class structure

<class> [Class]

The base class of all metaclasses, <class>, has the following slots. Note that these slots are for internal management, and users can’t change those information freely once the class is initialized.

It is recommended to obtain information about a class by procedures described in Section 7.2.3 [Class object], page 340, instead of directly accessing those slots.

name [Instance Variable of <class>]

The name of the class; the symbol given to define-class macro. class-name returns this value.
Chapter 7: Object system

The initialize method for `<class>`

```scheme
initialize (class `<class>`) initargs
```

The `define-class` macro expands into a call of `(make `<class>` ...)`, which allocates a class metaobject and calls `initialize` method. This method takes care of computing inheritance order (class precedence list) and calculate slots, and set up various internal slots. Then, at the very end of this method, it `freezes` the essential class slots; they became immutable.

Calculation of inheritance and slots are handle by generic functions. If you define a metaclass, you can define methods for them to customize how those calculations are done. Class inheritance is calculated by `compute-cpl` defined below. Slot calculation is a bit involved, and explained in the next subsection (see Section 7.5.2 [Customizing slot access], page 354).

If your class needs to initialize auxiliary slots, you can define your own `initialize` method on its metaclass, in which you call `next-method` first to set up the core part of the `<class>`...
structure, then you set up class-specific part. One caveat is that, after next-method handles initialization of the core <class> part, you can no longer modify essential class slots. If you need to tweak those slots, you can override class-post-initialize method, which is called right before the core class slots are frozen.

**compute-cpl class**  
This generic function is called from initialize method on <class>, and responsible to compute the class precedence list (CPL).

At the time this generic function is called, only name and direct-supers slots of class are set. The direct-supers slot contains a list of classes class directly inherits from. All classes in it is already initialized.

It must return a list of classes, starting with class itself and ending with <top>, representing the order of precedence with which methods are searched. The method defined for <class> uses C3 linearlization, which topologically sorts all the classes involved in the inheritance.

Override this method if you need to change how CPL is computed. You might not want to change the actual algorithm unless you emulate different object system, but you can use the method to ensure certain class is always inherited, for example.

**class-post-initialize class initargs**  
This generic function is called after all core initialization of class is finished, but before the class is “freezed”, that is, the essential parts of class becomes immutable. If you want to trick object system in some weird way, override this method.

We assume you know what you are doing, for object system assumes the essential parts are computed in the standard way. Messing with them can easily break the system.

### 7.5.2 Customizing slot access

**compute-slots class**  
**compute-get-n-set class slot-definition**  
These two generic functions are responsible to determine what slots a class has, and how each slot is accessed.

In the initialize method of a class, compute-slots is called after the class’s direct-supers, cpl and direct-slots are set. It must decide what slots the class should have, and what slot options each slot should have, based on those three piece of information.

The returned value should have the following form, and it is used as the value of the slots slot of the class.

```
<slots> : (<slot-definition> ...)  
<slot-definition> : (<slot-name> . <slot-options>)  
<slot-name> : symbol  
<slot-options> : keyword-value alternating list.
```

After the slots slot of the class is set by the returned value from compute-slots, compute-get-n-set is called for each slot to calculate how to access and modify the slot. The class and the slot definition are the arguments. It must return either one of the followings: an integer n

This slot becomes n-th instance slot. This is the only way to allocate a slot per instance.

The base method of compute-get-n-set keeps track of the current number of allocated instance slots in the class’s num-instance-slots slot. It is not recommended for other specialized methods to use or change the value of this slot, unless you know a very good reason to override the object system behavior in
deep down. Usually it is suffice to call next-method to let the base method reserve an instance slot for you.

See the examples below for modifying instance slot access behaviors.

a list (get-proc set-proc bound?-proc initializable)

The get-proc, set-proc and bound?-proc elements are procedures invoked when this slot of an instance is accessed (either via slot-ref/slot-set!/slot-bound?, or an accessor method specified by :getter/:setter slot options). The value other than get-proc may be #f, and can be omitted if all the values after it is also #f. That is, the simplest form of this type of return value is a list of one element, get-proc.

- When this slot is about to be read, get-proc is called with an argument, the instance. The returned value of get-proc is the value of the slot.
  
  The procedure may return #<undef> to indicate the slot is unbound. It triggers the slot-unbound generic function. (That is, this type of slot cannot have #<undef> as its value.)

- When this slot is about to be written, set-proc is called with two arguments, the instance and the new value. It is called purely for the side effect; the procedure may change the value of other slot of the instance, for example.
  
  If this element is #f or omitted, the slot becomes read-only; any attempt to write to the slot will raise an error.

- When slot-bound? is called to check whether the slot of an instance is bound, bound?-proc is called with an argument, the instance. It should return a boolean value which will be the result of slot-bound?.
  
  If this element is #f or omitted, slot-bound? will call get-proc and returns true if it returns #<undef>.

- The last element, initializable, is a flag that indicates whether this slot should be initialized when :init-value or :init-form.

A <slot-accessor> object

Access to this slot is redirected through the returned slot-accessor object. See below for more on <slot-accessor>.

The value returned by compute-get-n-set is immediately passed to compute-slot-accessor to create a slot accessor object, which encapsulates how to access and modify the slot.

After all slot definitions are processed by compute-get-n-set and compute-slot-accessor, an assoc list of slot names and <slot-accessor> objects are stored in the class’s accessors slot.

compute-slot-accessor [Generic Function]

compute-slot-accessor (class <class>) slot access-specifier [Method]

Access-specifier is a value returned from compute-get-n-set. The base method creates an instance of <slot-accessor> that encapsulates how to access the given slot.

Created slot accessor objects are stored (as an assoc list using slot names as keys) in the class’s accessors slot. Standard slot accessors and mutators, such as slot-ref, slot-set!, slot-bound?, and the slot accessor methods specified in :getter, :setter and :accessor slot options, all go through slot accessor object eventually. Specifically, those functions and methods first looks up the slot accessor object of the desired slot, then calls slot-ref-using-accessor etc.
compute-slots (class <class>)  [Method]
The standard method walks CPL of class and gathers all direct slots. If slots with the same
name are found, the one of a class closer to class in CPL takes precedence.

compute-get-n-set (class <class>) slot  [Method]
The standard processes the slot definition with the following slot allocations: :instance,
:class, each-subclass and :virtual.

slot-ref-using-accessor obj slot-accessor  [Function]
slot-set-using-accessor! obj slot-accessor value  [Function]
slot-bound-using-accessor? obj slot-accessor  [Function]
slot-initialize-using-accessor! obj slot-accessor initargs  [Function]

The low-level slot accessing mechanism. Every function or method that needs to read or
write to a slot eventually comes down to one of these functions.

Ordinary programs need not call these functions directly. If you ever need to call them, you
have to be careful not to grab the reference to slot-accessor too long; if obj’s class is changed
or redefined, slot-accessor can no longer be used.

Here we show a couple of small examples to illustrate how slot access protocol can be
customized. You can also look at gauche.mop.* modules (in the source tree, look under
lib/gauche/mop/) for more examples.

The first example implements the same functionality of :virtual slot allocation. We add

(define-class <procedural-slot-meta> (<class>) ())

(define-method compute-get-n-set ((class <procedural-slot-meta>) slot)
  (if (eqv? (slot-definition-allocation slot) :procedural)
    (let ([get-proc (slot-definition-option slot :ref)]
            [set-proc (slot-definition-option slot :set!)]
            [bound-proc (slot-definition-option slot :bound?)])
      (list get-proc set-proc bound-proc))
    (next-method)))

A specialized compute-get-n-set is defined on a metaclass <procedural-slot-meta>. It
checks the slot allocation, handles it if it is :procedural, and delegates other slot allocation
cases to next-method. This is a typical way to add new slot allocation by layering.

To use this :procedural slot, give <procedural-slot-meta> to a :metaclass argument of
define-class:

(define-class <temp> ()
  ((temp-c :init-keyword :temp-c :init-value 0)
   (temp-f :allocation :procedural
       :ref  (lambda (o) (+ (* (ref o 'temp-c) 9/5) 32))
       :set!  (lambda (o v)
             (set! (ref o 'temp-c) (* (- v 32) 5/9)))
       :bound? (lambda (o) (slot-bound? o 'temp-c))))
   :metaclass <procedural-slot-meta>)

An instance of <temp> keeps a temperature in both Celsius and Fahrenheit. Here’s an example
interaction.

gosh> (define T (make <temp>))
T

gosh> (d T)
<<temp> 0xb6b5c0> is an instance of class <temp>
Chapter 7: Object system

slots:
  temp-c : 0
  temp-f : 32.0

```
gosh> (set! (ref T 'temp-c) 100)
#<undef>
gosh> (d T)
#<<temp> 0xb6b5c0> is an instance of class <temp>
slots:
  temp-c : 100
  temp-f : 212.0
```

```
gosh> (set! (ref T 'temp-f) 450)
#<undef>
gosh> (d T)
#<<temp> 0xb6b5c0> is an instance of class <temp>
slots:
  temp-c : 232.22222222222223
  temp-f : 450.0
```

Our next example is a simpler version of `gauche.mop.validator`. We add a slot option :filter, which takes a procedure that is applied to a value to be set to the slot.

```
(define-class <filter-meta> (<class>) ()
)
```

```
(define-method compute-get-n-set ((class <filter-meta>) slot)
  (cond [(slot-definition-option slot :filter #f)
            => (lambda (f)
                (let1 acc (compute-slot-accessor class slot (next-method))
                  (list (lambda (o) (slot-ref-using-accessor o acc))
                        (lambda (o v) (slot-set-using-accessor! o acc (f v)))
                        (lambda (o) (slot-bound-using-accessor? o acc))
                        #t)))]
      [else (next-method)]))
```

The trick here is to call `next-method` and `compute-slot-accessor` to calculate the slot accessor and wrap it. See how this metaclass works:

```
(define-class <foo> ()
  ((v :init-value 0 :filter x->number))
  :metaclass <filter-meta>)
```

```
gosh> (define foo (make <foo>))
foo
gosh> (ref foo'v)
0
gosh> (set! (ref foo'v) "123")
#<undef>
gosh> (ref foo'v)
123
```

### 7.5.3 Method instantiation

```
make (class <method>) :rest initargs
```

### 7.5.4 Customizing method application

```
apply-generic gf args
```
sort-applicable-methods  gf methods args  [Generic Function]
method-more-specific?  method1 method2 classes  [Generic Function]
apply-methods  gf methods args  [Generic Function]
apply-method  gf method build-next args  [Generic Function]

7.5.5 Customizing class redefinition

class-redefinition  old-class new-class  [Generic Function]

When a class is redefined (see Section 7.2.5 [Class redefinition], page 342), a new class metaobject is instantiated by (make metaclass initargs ...), then this generic function is called with the old class metaobject and new class metaobject. It should transform the information in the old class into the new class.

The default method, (class-redefinition <class> <class>), takes care of updating all the methods referencing to the old class, and propagate changes to the superclasses and subclasses. If you customize this generic function, you should call the default method with next-method to make sure those basic bookkeeping is done, or unexpected things can happen.

Class redefinition mutates lots of structures. If you throw an error in middle of it, the internal state can be left inconsistent.

Internally, the system uses a single mutex dedicated for the class redefinition so that only one thread can execute it at a time. You don’t need to worry about other thread stepping on during class-redefinition method (Other thread can still be running for other operations, though, so if you touch objects that can be touched from outside of class redefinition, you should mutex it.)
8 Library modules - Overview

In the following chapters, we explain library modules bundled with Gauche’s distribution. These modules should generally be loaded and imported (usually using \texttt{use} - See Section 4.13.4 [Using modules], page 81, for details), unless otherwise noted.

Some modules are described as "autoloaded". That means you don’t need to \texttt{load} or \texttt{use} the module explicitly; at the first time the bindings are used in the program, the module is automatically loaded and imported. See Section 6.22.4 [Autoload], page 289, for the details of autoloading.

As the number of bundled libraries grows, it becomes harder to find the one you need. If you feel lost, check out the section Section 8.1 [Finding libraries you need], page 359, in which we categorize libraries by their purposes.

The following four chapters describe bundled modules, grouped by their names.

- Chapter 9 [Library modules - Gauche extensions], page 366, contains a description of \texttt{gauche.*} modules, which are more or less considered the core features of Gauche but separated since less frequently used. (Some modules are rather ad-hoc, but here for historical reasons).

- Chapter 10 [Library modules - R7RS standard libraries], page 580, explains how Gauche integrates R7RS into existing Gauche structures. If you want to write R7RS-compliant portable programs, you definitely want to check the first two sections of this chapter. What follows is the description of R7RS modules. The “small” part of R7RS has been frozen, but the “large” part—additional libraries—are still growing.

- Chapter 11 [Library modules - SRFIs], page 689, describes the modules which provide SRFI functionalities. They have the names beginning with \texttt{srfi}. Note that some of SRFI features are built in Gauche core and not listed here. See Section 2.1 [Standard conformance], page 5, for the entire list of supported SRFIs.

- Chapter 12 [Library modules - Utilities], page 808, describes other modules—including database interface, filesystem utilities, network protocol utilities, and more.

There are a few procedures that help your program to check the existence of certain modules or libraries at run-time. See Section 6.22.5 [Operations on libraries], page 289, for the details.

8.1 Finding libraries you need

Each module is named more or less after what it implements rather than what it is implemented \textit{for}. If the module solves one problem, both are the same. However, sometimes there are multiple ways to solve a problem, or one implementation of an algorithm can solve multiple different problems; thus it is difficult to name the modules in problem-oriented (or purpose-oriented) way.

Because of this, it may not be straightforward for a newcomer to Gauche to find an appropriate Gauche module to solve her problem, since there may be multiple algorithms to do the job, and each algorithm can be implemented in different modules.

The modules are also designed in layers; some low-level modules provide direct interface to the system calls, while some higher-level ones provide more abstract, easy-to-use interface, possibly built on top of more than one low-level modules. Which one should you use? Generally you want to use the highest level, for the very purpose of libraries are to provide easy, abstract interface. However there are times that you have to break the abstraction and to go down to tweak the machinery in the basement; then you need to use low-level modules directly.

The purpose of this section is to group the libraries by their purposes. Each category lists relevant modules with brief descriptions.
8.1.1 Library directory - data containers

Generic container operations

Some data containers have similar properties; for example, lists, vectors and hash tables can be seen as a collection of data. So it is handy to have generic operators, such as applying a procedure to all the elements.

Gauche provides such mechanism to a certain degree, mainly using its object system.

- **Collection** - Generic functions applicable for unordered set of values. See Section 9.6 [Collection framework], page 404.
- **Sequence** - Generic functions applicable for ordered set of values. See Section 9.31 [Sequence framework], page 510.
- **Dictionary** - Generic functions to handle dictionary, that is, a mapping from keys to values. See Section 9.10 [Dictionary framework], page 428.
- **Relation** - Generic functions to handle relations (in a sense of Codd’s definition). See Section 12.89 [Relation framework], page 1031.
- **Comprehension** - This is a collection of macros very handy to construct and traverse collections/sequences in concise code. See Section 11.11 [Eager comprehensions], page 711.

Container implementations

- **List** - the universal data structure. You want to check Section 6.6 [Pairs and lists], page 140, and Section 10.3.1 [R7RS lists], page 593.
- **Vector** - a one-dimensional array of arbitrary Scheme values. See Section 6.13.1 [Vectors], page 195, and Section 10.3.2 [R7RS vectors], page 597. If you need a wide range of index, but the actual data is sparse, you might want to look at Section 12.24.1 [Sparse vectors], page 854.
- **Uniform vector** - a special kind of vectors that can hold limited types of values (e.g. integers representable in 8bits). It tends to be used in performance sensitive applications, such as graphics. See Section 6.13.2 [Uniform vectors], page 198.
- **Array** - multi-dimensional arrays that can hold arbitrary Scheme values. See Section 9.1 [Arrays], page 366.
- **Uniform array** - multi-dimensional arrays that can hold limited types of values. This is also supported by Section 9.1 [Arrays], page 366.
- **String** - a sequence of characters. See Section 6.11 [Strings], page 170, and Section 11.6 [String library], page 692. Gauche handles multibyte strings — see Section 2.2 [Multibyte strings], page 13, for the details.
- **Character set** - a set of characters. See Section 6.10 [Character sets], page 165, and Section 10.3.6 [R7RS character sets], page 614.
- **Hash table** - hash tables. See Section 6.14.1 [Hashtables], page 206. For very large hash tables (millions of entries), Section 12.24.3 [Sparse tables], page 858, may provide better memory footprint.
- **Balanced tree** - If you need to order keys in a dictionary, you can use treemaps. See Section 6.14.2 [Treemaps], page 212.
- **Immutable map** - Sometimes immutable dictionary is handy. Internally it implements a functional balanced tree. See Section 12.17 [Immutable map], page 834.
- **Queue** - Both fast and thread-safe queues are provided in Section 12.19 [Queue], page 837. Thread-safe queues can also be used as synchronized messaging channel.
- **Heap** - See Section 12.15 [Heap], page 831.
- **Ring buffer** - Space-efficient ring buffer. See Section 12.22 [Ring buffer], page 849.
• **Cache** - Various cache algorithm implementations. See Section 12.14 [Cache], page 828.

• **Record** - a simple data structure. Although Gauche’s object system can be used to define arbitrary data structures, you might want to look at Section 9.28 [Record types], page 502, and Section 12.88 [SLIB-compatible record type], page 1029, for they are more portable and potentially more efficient.

• **Stream** - you can implement cool lazy algorithms with it. See Section 12.90 [Stream library], page 1033.

• **Trie** - Another tree structure for efficient common-prefix search. See Section 12.25 [Trie], page 859.

• **Database interface** - dbm interface can be used as a persistent hash table; see Section 12.28 [Generic DBM interface], page 871. For generic RDBMS interface, see Section 12.27 [Database independent access layer], page 865.

### 8.1.2 Library directory - string and character

Basic string operations are covered in Section 6.11 [Strings], page 170, and Section 11.6 [String library], page 692. A string is also a sequence of characters, so you can apply methods in Section 9.6 [Collection framework], page 404, and Section 9.31 [Sequence framework], page 510.

Character and character set operations are covered in Section 6.9 [Characters], page 160, Section 6.10 [Character sets], page 165, and Section 10.3.6 [R7RS character sets], page 614.

If you scan or build strings sequentially, do not use index access. String ports (see Section 6.21.5 [String ports], page 268) provides more efficient, and elegant way.

You can use regular expressions to search and extract character sequences from strings; see Section 6.12 [Regular expressions], page 183.

If you need to deal with low-level (i.e. byte-level) representation of strings, Section 6.13.2 [Uniform vectors], page 198, has some tools to convert strings and byte vectors back and forth.

Are you dealing with a structure higher than a mere sequence of characters? Then take a look at **text.*** modules. Section 12.74 [Parsing input stream], page 1007, has some basic scanners. Section 12.79 [Transliterate characters], page 1013, implements a feature similar to Unix’s `tr(1)`. You can take `diff` of two texts; see Section 12.65 [Calculate difference of text streams], page 991. And if you want to construct large text from string fragments, do not use `string-append`—see Section 12.80 [Lazy text construction], page 1014.

Last but not least, Gauche has support of various character encoding schemes. See Section 9.5 [Character code conversion], page 399, for the basic utilities. Most higher-level functions such as `open-input-file` can take `:encoding` keyword argument to perform character conversion implicitly. Also see Section 2.3 [Multibyte scripts], page 14, if you write Scheme program in non-ASCII characters. If you want to process Gauche source code which may contain "encoding" magic comment, see Section 6.21.6 [Coding-aware ports], page 270. Gauche also has GNU gettext compatible module (Section 12.69 [Localized messages], page 1000) if you need localization.

### 8.1.3 Library directory - data exchange

Most useful programs need to communicate with outside world (other programs or humans). That involves reading the external data into your program understanding whatever format the data is in, and also writing the data in the format the others can understand.

Lots of network-related external formats are defined in RFC, and there are corresponding **rfc.*** module that handle some of them. See Section 12.41 [RFC822 message parsing], page 917, for example, to handle the pervasive RFC2822 message format. Or, JSON can be handled by Section 12.49 [JSON parsing and construction], page 937.

When you exchange table-formatted data, one of the easiest way may be the plain text, one row per line, and columns are separated by some specific characters (e.g. comma). See Section 12.64 [CSV tables], page 988, for basic parser/writer for them.
Oh, and nowadays every business user wants XML, right? You know they are just S-expressions with extra redundancy and pointy parentheses. So why don’t you read XML as if they’re S-exprs, process them with familiar cars and cdrs and maps, then write them out with extra redundancy and pointy parentheses? Module sxml.ssax (Section 12.59 [Functional XML parser], page 959) implements SAX XML parser, with which you can parse XML and process them on the fly, or convert it to SXML, S-expression XML. You can query SXML using SXPath, an XPath counterparts of S-expression (Section 12.60 [SXML query language], page 970). You can output all kinds of XML and HTML using the SXML serializer (Section 12.62 [Serializing XML and HTML from SXML], page 983).

(But you know most web services nowadays also talks JSON, and that’s much lighter and handier than XML. See Section 12.49 [JSON parsing and construction], page 937.)

It is planned that various file format handling routines would be available as file.* modules, though we have none ready yet. If you plan to write one, please go ahead and let us know!

8.1.4 Library directory - files

Files and directories. Roughly speaking, there are two places you want to look at.

Section 6.24.4 [Filesystems], page 296, in the core, has routines close to the underlying OS provides. If you have experience with Unix system programming you’ll find familiar function names there. The fcntl functionality is split to gauche.fcntl (Section 9.11 [Low-level file operations], page 432), FYI.

Also you definitely want to look at file.util (Section 12.34 [Filesystem utilities], page 881), which implements higher-level routines on top of system-level ones.

8.1.5 Library directory - processes and threads

Process-related routines also come in two levels.

The gauche.process module provides high-level routines (Section 9.27 [High-level process interface], page 488); you can pipe the data into and out of child processes easily, for example.

Gauche core provides the primitive fork and exec interface as well as the convenient system call (see Section 6.24.10 [Process management], page 318). Use them when you want a precise control over what you’re doing.

Gauche has preemptive threads on most Unix platforms including OSX. Check out Section 9.35 [Threads], page 530, for the basic thread support, including primitive mutexes. The data.queue module (see Section 12.19 [Queue], page 837) provides thread-safe queue that can also be handy for synchronization. Thread pool is available in control.thread-pool (see Section 12.12 [Thread pools], page 826).

8.1.6 Library directory - networking

We have multi-layer abstraction here. At the bottom, we have APIs corresponding to socket-level system calls. In the middle, a convenience library that automates host name lookups, connection and shutdown, etc. On top of them we have several modules that handles specific protocols (e.g. http).

The gauche.net module (Section 9.22 [Networking], page 464) provides the bottom and middle layer. For the top layer, look for rfc.* modules, e.g. rfc.http (Section 12.46 [HTTP client], page 930). More protocol support is coming (there are rfc.ftp and rfc.imap4 written by users, which are waiting for being integrated into Gauche—maybe in next release).

There’s a plan of even higher level of libraries, under the name net.*, which will abstract more than one network protocols. The planned ones include sending emails, or universal resource access by uri. Code contributions are welcome.

8.1.7 Library directory - input and output
8.1.8 Library directory - time

8.1.9 Library directory - bits and bytes

Binary I/O

As the bottom level, Gauche includes primitive byte I/O \(\text{read-byte, write-byte}\) as well as block I/O \(\text{read-uvector, read-uvector!, write-uvector}\) in its core. (See Section 6.21.7.1 \[Reading data\], page 270, Section 6.21.8 \[Output\], page 275, and Section 9.38.4 \[Uvector block I/O\], page 567).

As the middle level, the module \texttt{binary.io} (Section 12.1 \[Binary I/O\], page 808) has routines to retrieve specific datatype with optional endian specification.

And as the top level, the module \texttt{binary.pack} (Section 12.2 \[Packing binary data\], page 811) allows packing and unpacking structured binary data, a la Perl’s \texttt{pack/unpack}.

Bit manipulation

Gauche core provides basic bitshift and mask operations (see Section 6.3.6 \[Basic bitwise operations\], page 136). SRFI-151 has comprehensive bitwise operations (see Section 10.3.22 \[R7RS bitwise operations\], page 664).

8.2 Naming convention of libraries

The following table summarizes naming categories of the modules, including external ones and planned ones.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary.*</td>
<td>Utilities to treat binary data</td>
</tr>
<tr>
<td>compat.*</td>
<td>Provides compatibility layers</td>
</tr>
<tr>
<td>control.*</td>
<td>Control structures, such as multithreading utilities</td>
</tr>
<tr>
<td>data.*</td>
<td>Implementations of various data structures</td>
</tr>
<tr>
<td>dbi.<em>, dbd.</em></td>
<td>Database independent interface layer and drivers.</td>
</tr>
<tr>
<td>dbm.*</td>
<td>DBM interface</td>
</tr>
<tr>
<td>gauche.*</td>
<td>Stuffs more or less considered as Gauche core features.</td>
</tr>
<tr>
<td>gl.*</td>
<td>OpenGL binding and related libraries (external package).</td>
</tr>
<tr>
<td>gtk.*</td>
<td>GTK+ binding and related libraries (external package).</td>
</tr>
<tr>
<td>file.*</td>
<td>Manipulating files and directories</td>
</tr>
<tr>
<td>lang.*</td>
<td>Language-related libraries, artificial and/or natural (planned).</td>
</tr>
<tr>
<td>math.*</td>
<td>Mathematics</td>
</tr>
<tr>
<td>os.*</td>
<td>Features for specific OSes</td>
</tr>
<tr>
<td>parser.*</td>
<td>Parsers</td>
</tr>
<tr>
<td>rfc.*</td>
<td>Implementations of net protocols defined in RFC’s.</td>
</tr>
<tr>
<td>scheme.*</td>
<td>R7RS libraries</td>
</tr>
<tr>
<td>srfi.*</td>
<td>SRFI implementations</td>
</tr>
<tr>
<td>sxml.*</td>
<td>SXML libraries</td>
</tr>
</tbody>
</table>
text.* Libraries dealing with text data.
util.* Generic implementations of various algorithms.
www.* Implementations of various protocols and formats mainly used in WWW.

8.3 Obsolete and superseded modules
During the course of development of Gauche, some modules have been renamed, merged, or dissolved into the core. Also, some SRFI libraries become standard and given a new name, or superseded with a newer SRFI library.

We list such modules here for the reference. New code shouldn’t use these modules, although they are kept in the distribution so that legacy code can keep running.

Obsolete modules
text.unicode [Module]
Renamed to gauche.unicode. See Section 9.37 [Unicode utilities], page 549.
util.list [Module]
Dissolved into the core. No longer needed.
util.queue [Module]
Renamed to data.queue. See Section 12.19 [Queue], page 837.
util.rbtree [Module]
Incorporated into the core as built-in object <tree-map>. See Section 6.14.2 [Treemaps], page 212.
The following procedures are aliases of the ones with replacing rbtree for tree-map, e.g. rbtree-get is the same as tree-map-get.

```
make-rbtree  rbtree?  rbtree-get  rbtree-put!
rbtree-delete! rbtree-exists?  rbtree-empty?  rbtree-update!
rbtree-push!  rbtree-pop!  rbtree-num-entries  rbtree->alist
alist->rbtree  rbtree-keys  rbtree-values  rbtree-copy
rbtree-fold  rbtree-fold-right
```
The following procedures are similar to tree-map-min, tree-map-max, tree-map-pop-min! and tree-map-pop-max!, respectively, except that the rbtree-* version takes an optional default argument and returns it when the tree is empty, and raise an error if no default argument is provided and tree is empty. (The tree-map version just returns #f for the empty tree.)

```
rbtree-min  rbtree-max
rbtree-extract-min!  rbtree-extract-max!
```
The following procedure doesn’t have corresponding API in tree-map. It checks internal consistency of the given tree-map.

```
rbtree-check
```
util.sparse [Module]
Renamed to data.sparse. See Section 12.24 [Sparse data containers], page 853.
util.trie [Module]
Renamed to data.trie. See Section 12.25 [Trie], page 859.
Superseded modules

**srfi.1** [Module]
SRFI-1 (List library) has become a part of R7RS large, as `scheme.list`. See Section 10.3.1 [R7RS lists], page 593.

**srfi.14** [Module]
SRFI-14 (Character-set library) has become a part of R7RS large, as `scheme.charset`. See Section 10.3.6 [R7RS character sets], page 614.

**srfi.43** [Module]
Vector library (Legacy) - this module is effectively superseded by R7RS and `srfi.133`. See Section 6.13.1 [Vectors], page 195, and see Section 10.3.2 [R7RS vectors], page 597.

**srfi.60** [Module]
Integers as bits - this module is superseded by `srfi.151`. See Section 10.3.22 [R7RS bitwise operations], page 664.

**srfi.69** [Module]
Basic hash tables - this module is superseded by R7RS `scheme.hash-table`. See Section 10.3.7 [R7RS hash tables], page 618.

**srfi.111** [Module]
SRFI-111 (Boxes) has become a part of R7RS `scheme.box` module. See Section 10.3.15 [R7RS boxes], page 636.

**srfi.113** [Module]
SRFI-113 (Sets and bags) has become a part of R7RS `scheme.set`. See Section 10.3.5 [R7RS sets], page 606.

**srfi.114** [Module]
Comparators - R7RS favored SRFI-128 over this srfi to make `scheme.comparator` (Section 10.3.18 [R7RS comparators], page 640), so adoption of this srfi may not be as wide. Note that, in Gauche, a native comparator object can be used for SRFI-114 procedures, and this module provides some useful additional utilities. It’s ok to use this module if portability isn’t a big issue.

**srfi.117** [Module]
SRFI-117 has become R7RS’s `scheme.list-queue`. See Section 10.3.16 [R7RS list queues], page 636.

**srfi.127** [Module]
SRFI-127 has become R7RS’s `scheme.lseq`. See Section 10.3.13 [R7RS lazy sequences], page 633.

**srfi.132** [Module]
SRFI-132 has become R7RS’s `scheme.sort`. See Section 10.3.4 [R7RS sort], page 602.

**srfi.133** [Module]
SRFI-133 has become R7RS’s `scheme.vector`. See Section 10.3.2 [R7RS vectors], page 597.
9 Library modules - Gauche extensions

9.1 gauche.array - Arrays

gauche.array

This module provides multi-dimensional array data type and operations. The primitive API follows SRFI-25. Besides a generic SRFI-25 array that can store any Scheme objects, this module also provides array classes that stores numeric objects efficiently, backed up by homogeneous numeric vectors (see Section 6.13.2 [Uniform vectors], page 198). An external representation of arrays, using SRFI-10 mechanism, is also provided.

Each element of an N-dimensional array can be accessed by N integer indices, \([i_0 \ i_1 \ldots \ i_{N-1}]\). An array has associated shape that knows lower-bound \(s_k\) and upper-bound \(e_k\) of index of each dimension, where \(s_k \leq e_k\), and the index \(i_k\) must satisfy \(s_k \leq i_k < e_k\). (Note: it is allowed to have \(s_k == e_k\), but such array can’t store any data. It is also allowed to have zero-dimensional array, that can store a single data.). The shape itself is a \([D \times 2]\) array, where \(D\) is the dimension of the array which the shape represents.

You can pass index(es) to array access primitives in a few ways; each index can be passed as individual argument, or can be ‘packed’ in a vector or one-dimensional array. In the latter case, such a vector or an array is called an "index object". Using a vector is efficient in Gauche when you iterate over the elements by changing the vector elements, for it won’t involve memory allocation.

Arrays can be compared by the \texttt{equal?}\ procedure. \texttt{Equal?} returns \#t if two arrays have the same shape and their corresponding elements are the same in the sense of \texttt{equal?}.

Internally, an array consists of a backing storage and a mapping procedure. A backing storage is an object of aggregate type that can be accessed by an integer index. A mapping procedure takes multi-dimensional indices (or index object) and returns a scalar index into the backing storage.

\texttt{<array-base>}

\{gauche.array\} An abstract base class of array types, that implements generic operations on the array. To create an array instance, you should use one of the following concrete array classes.

\texttt{<array>}
\texttt{<u8array>}
\texttt{<s8array>}
\texttt{<u16array>}
\texttt{<s16array>}
\texttt{<u32array>}
\texttt{<s32array>}
\texttt{<u64array>}
\texttt{<s64array>}
\texttt{<f16array>}
\texttt{<f32array>}
\texttt{<f64array>}

\{gauche.array\} Concrete array classes. The \texttt{<array>} class implements SRFI-25 compatible array, i.e. an array that can store any Scheme objects. The \texttt{<u8array>} class through \texttt{<f64array>} classes uses a \texttt{<u8vector>} through \texttt{<f64vector>} as a backing storage, and can only store a limited range of integers or inexact real numbers, but they are space efficient.
An array is written out in this format. (Substitute \texttt{<array>} for \texttt{<u8array>}, etc.) \texttt{shape} is a list of even number of integers, and each $2n$-th integer and $2n+1$-th integer specifies the inclusive lower-bound and exclusive upper-bound of $n$-th dimension, respectively. The following \texttt{obj} \texttt{...} are the values in the array listed in row-major order.

When read back, this syntax is read as an array with the same shape and content, so it is equal to the original array.

\begin{verbatim}
; an array such that:
;  8 3 4
;  1 5 9
;  6 7 2
#,(<array> (0 3 0 3) 8 3 4 1 5 9 6 7 2)
\end{verbatim}

; a 4x4 identity matrix
\begin{verbatim}
#,(<array> (0 4 0 4) 1 0 0 0 1 0 0 0 1 0 0 0 1)
\end{verbatim}

\begin{description}
\item{array? obj} \hfill [Function]
\end{description}

\begin{verbatim}
[SRFI-25]{gauche.array} Returns \texttt{#t} if \texttt{obj} is an array, \texttt{#f} otherwise. It is equivalent to (is-a? obj <array-base>).
\end{verbatim}

\begin{description}
\item{make-array shape :optional init} \hfill [Function]
\end{description}

\begin{verbatim}
[SRFI-25]{gauche.array} Creates an array of shape \texttt{shape}. \texttt{Shape} must be a \texttt{[D x 2]} array, and for each $k$ ($0 \leq k < D$), the \texttt{[k 0]} element must be less than or equal to the \texttt{[k 1]} element. If \texttt{init} is given, all the elements are initialized by it. Otherwise, the initial value of the elements are undefined.
\end{verbatim}

\begin{verbatim}
(make-array (shape 0 2 0 2 0 2) 5)
⇒ #,(<array> (0 2 0 2 0 2) 5 5 5 5 5 5)
\end{verbatim}

\begin{description}
\item{make-u8array shape :optional init} \hfill [Function]
\item{make-s8array shape :optional init} \hfill [Function]
\item{make-f32array shape :optional init} \hfill [Function]
\item{make-f64array shape :optional init} \hfill [Function]
\end{description}

\begin{verbatim}
{gauche.array} Like \texttt{make-array}, but creates and returns an uniform numeric array.
\end{verbatim}

\begin{description}
\item{array-copy array} \hfill [Function]
\end{description}

\begin{verbatim}
{gauche.array} Returns a copy of \texttt{array}, with the same class, shape and content.
\end{verbatim}

\begin{description}
\item{shape bound \texttt{...}} \hfill [Function]
\end{description}

\begin{verbatim}
[SRFI-25]{gauche.array} Takes even number of exact integer arguments, and returns a two-dimensional array that is suitable for representing the shape of an array.
\end{verbatim}

\begin{verbatim}
(shape 0 2 1 3 3 5)
⇒ #,(<array> (0 3 0 2) 0 2 1 3 3 5)
\end{verbatim}

\begin{verbatim}
(shape)
⇒ #,(<array> (0 0 0 2))
\end{verbatim}

\begin{description}
\item{array shape init \texttt{...}} \hfill [Function]
\end{description}

\begin{verbatim}
[SRFI-25]{gauche.array} Creates an array of shape \texttt{shape}, initializing its elements by \texttt{init}.
\end{verbatim}

\begin{verbatim}
(array (shape 0 2 1 3) 'a 'b 'c 'd)
⇒ #,(<array> (0 2 1 3) a b c d)
\end{verbatim}
Chapter 9: Library modules - Gauche extensions

`u8array shape init . . .`  
`s8array shape init . . .`  

`f32array shape init . . .`  
`f64array shape init . . .`  

{gauche.array} Like array, but creates and returns an uniform numeric array initialized by init . . .

```
(u8array (shape 0 2 0 2) 1 2 3 4)  
⇒ #,(<u8array> (0 2 0 2) 1 2 3 4)
```

`array-rank array`  
{SRFI-25}      
{gauche.array} Returns the number of dimensions of an array array.

```
(array-rank (make-array (shape 0 2 0 2))) ⇒ 3
(array-rank (make-array (shape))) ⇒ 0
```

`array-shape array`  
{gauche.array} Returns a shape array of array.

`array-start array dim`  
`array-end array dim`  
`array-length array dim`  
{SRFI-25}      
{gauche.array} Array-start returns the inclusive lower bound of index of dim-th dimension of an array array. Array-end returns the exclusive upper bound. And array-length returns the difference between two. Array-start and array-end are defined in SRFI-25.

```
(define a (make-array (shape 1 5 0 2)))

(array-start a 0) ⇒ 1
(array-end a 0) ⇒ 5
(array-length a 0) ⇒ 4
(array-start a 1) ⇒ 0
(array-end a 1) ⇒ 2
(array-length a 1) ⇒ 2
```

`array-size array`  
{gauche.array} Returns the total number of elements in the array array.

```
(array-size (make-array (shape 5 9 1 3))) ⇒ 8
(array-size (make-array (shape))) ⇒ 1
(array-size (make-array (shape 0 0 0 2))) ⇒ 0
```

`array-ref array k . . .`  
`array-ref array index`  
{SRFI-25}      
{gauche.array} Gets the element of array array. In the first form, the element is specified by indices k . . . In the second form, the element is specified by an index object index, which must be a vector or an one-dimensional array.

`array-set! array k . . . value`  
`array-set! array index value`  
{SRFI-25}      
{gauche.array} Sets the element of array array to value. In the first form, the element is specified by indices k . . . In the second form, the element is specified by an index object index, which must be a vector or an one-dimensional array.
share-array array shape proc

[SRFI-25][gauche.array] creates and returns a new array of shape shape, that shares the backing storage with the given array array. The procedure proc maps the indices of the new array to the indices of the original array, i.e. proc must be a n-ary procedure that returns m values, where n is the dimension of the new array and m is the one of the original array. Furthermore, proc must be an affine function; each mapping has to be a linear combination of input arguments plus optional constant. (Share-array optimizes the mapping function based on the affinity assumption, so proc won’t be called every time the new array is accessed).

array-for-each-index array proc :optional index

{gauche.array} calls proc with every index of array. If no index argument is provided, proc is called as (proc i j k ...), in which (i,j,k,...) walks over the index. It begins from the least index value of each dimension, and latter dimension is incremented faster.

gosh> (define a (array (shape 0 2 0 2) 1 2 3 4))

a

(, (<array> (0 2 0 2) 1 2 3 4))

(, (array-for-each-index a (^(i j) (print i","j))))

0,0

0,1

1,0

1,1

This form of passing indexes is simple but not very efficient, though. For better performance, you can pass an index object to an optional argument index, which is modified for each index and passed to proc. The index object must be mutable, and either a vector, an one-dimensional array, an s8vector, an s16vector or an s32vector. The length of the index object must match the rank of the array. Using index object is efficient since the loop won’t allocate. Don’t forget that the index object is destructively modified within the loop.

gosh> (array-for-each-index a (cut format #t "~s\n" <>)(vector 0 0))

#(0 0)

#(0 1)

#(1 0)

#(1 1)

gosh> (array-for-each-index a (cut format #t "~s\n" <>)(s8vector 0 0))

#s8(0 0)

#s8(0 1)

#s8(1 0)

#s8(1 1)

The procedure returns an unspecified value.

shape-for-each shape proc :optional index

{gauche.array} calls proc with all possible indexes represented by the shape shape. The optional index argument works the same way as array-for-each-index. Returns an unspecified value.

gosh> (shape-for-each (shape 0 2 0 2) (^(i j) (print i","j))))

0,0

0,1

1,0

1,1
tabulate-array shape proc :optional index

{gauche.array} Calls proc over each index represented by the shape shape, and creates an array from the result of proc. The optional index object can be used in the same way as array-for-each-index. The following example creates an identity matrix of the given shape:

(tabulate-array (shape 0 3 0 3) (^i j (if (= i j) 1 0)))
⇒ #,(<array> (0 3 0 3) 1 0 0 0 1 0 0 0 1)

array-retabulate! array proc :optional index
array-retabulate! array shape proc :optional index

{gauche.array} Calls proc over each index of the given array, and modifies the array’s element by the returned value of proc. The optional index object can be used in the same way as array-for-each-index. The second form takes a shape; it must match the array’s shape. It is redundant, but may allow some optimization in future in case shape is a literal. Returns an unspecified value.

array-map proc array0 array1 ...
array-map shape proc array0 array1 ...

{gauche.array} The arguments array0, array1, ... must be arrays with the same shape. For each set of corresponding elements of the input arrays, proc is called, and a new array of the same shape is created by the returned values. The second form takes a shape argument, which must match the shape of input array(s). It is redundant, but may allow some optimization in future in case shape is a literal.

(array-map - (array (shape 0 2 0 2) 1 2 3 4))
⇒ #,(<array> (0 2 0 2) -1 -2 -3 -4)

array-map! array proc array0 array1 ...
array-map! array shape proc array0 array1 ...

{gauche.array} Like array-map, but the results of proc are stored by the given array, whose shape must match the shape of input array(s). Returns unspecified value.

array->vector array
array->list array

{gauche.array} Returns a fresh vector or a fresh list of all elements in array.

(array->vector
  (tabulate-array (shape 1 3 1 4) (^i j (+ (* 10 i) j))))
⇒ #(11 12 13 21 22 23)

array-concatenate a b :optional dimension

{gauche.array} Concatenates arrays at the specified dimension. The sizes of the specified dimension of two arrays must match, although the shapes can be different. Arrays can be of any ranks, but two ranks must match.

;; [a b] [a b]
;; [c d] (+) => [c d]
;; [e f] [e f]
(array-concatenate
  (array (shape 0 2 0 2) 'a 'b 'c 'd)
  (array (shape 0 1 0 2) 'e 'f))
⇒ #,(<array> (0 3 0 2) a b c d e f)

;; [a b] [e] [a b e]
;; [c d] (+) [f] => [c d f]
(array-concatenate
  (array (shape 0 2 0 2) 'a 'b 'c 'd)
  (array (shape 0 2 0 1) 'e 'f)
  1)
⇒ #,(<array> (0 2 0 3) a b e c d f)

;; The index range can differ, as far as the sizes match
(array-concatenate
  (array (shape 0 2 0 2) 'a 'b 'c 'd)
  (array (shape 1 3 0 1) 'e 'f) 1)
⇒ #,(<array> (0 2 0 3) a b e c d f)

array-transpose array :optional dim1 dim2
{gauche.array} The given array must have a rank greater than or equal to 2. Transpose
the array’s dim1-th dimension and dim2-th dimension. The default is 0 and 1.

(array-rotate-90 array :optional dim1 dim2)
{gauche.array} The given array must have a rank greater than or equal to 2. We regard
the array as a matrix with dim1-th dimension as rows and dim2-th dimension as columns,
and returns a fresh array whose content is filled by rotating array 90 degree clockwise. The
defaults of dim1 and dim2 are 0 and 1, respectively.

;; [1 2 3]
;; [4 5 6] => [5 2]
;; [6 3]
(array-rotate-90 (array (shape 0 2 0 3) 1 2 3 4 5 6))
⇒ #,(<array> (0 3 0 2) 4 1 5 2 6 3)

If array has a rank greater than 2, the array is treated as a matrix of subarrays.

array-flip array :optional dimension
array-flip! array :optional dimension
{gauche.array} Flips the content of the array across the dimension-th dimension. (default
is 0). array-flip! modifies the content of array and return it. array-flip doesn’t modify
array but creates a fresh array with the flipped content and returns it.

;; [1 2 3] => [4 5 6]
;; [4 5 6] => [1 2 3]
(array-flip (array (shape 0 2 0 3) 1 2 3 4 5 6))
⇒ #,(<array> (0 2 0 3) 4 5 6 1 2 3)

;; [1 2 3] => [3 2 1]
;; [4 5 6] => [6 5 4]
(array-flip (array (shape 0 2 0 3) 1 2 3 4 5 6) 1)
⇒ #,(<array> (0 2 0 3) 3 2 1 6 5 4)

identity-array dimension :optional class
{gauche.array} Returns a fresh identity array of rank 2, with the given dimension. You can
pass one of array classes to class to make the result the instance of the class; the default class
is <array>.

(identity-array 3)
⇒ #,(<array> (0 3 0 3) 1 0 0 0 1 0 0 0 1)

(identity-array 3 <f32array>)
⇒ #,(<f32array> (0 3 0 3) 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0)
array-inverse array
{gauche.array} Regards the array as a matrix, and returns its inverse matrix; array must be 2-dimensional, and must have square shape. If array doesn’t satisfy these conditions, an error is thrown.

If array isn’t a regular matrix, #f is returned.

determinant array

determinant! array
{gauche.array} Regards the array as a matrix, and calculates its determinant; array must be 2-dimensional, and must have square shape. If array doesn’t satisfy these conditions, an error is thrown.

determinant! destructively modifies the given array during calculation. It is faster than determinant, which copies array before calculation to preserve it.

array-mul a b
{gauche.array} Arrays a and b must be rank 2. Regarding them as matrices, multiply them together. The number of rows of a and the number of columns of b must match.

;; [6 5] ;; [1 2 3] x [4 3] => [20 14] ;; [4 5 6] [2 1] [56 41]

(array-mul (array (shape 0 2 0 3) 1 2 3 4 5 6) (array (shape 0 3 0 2) 6 5 4 3 2 1))
⇒ #,(<array> (0 2 0 2) 20 14 56 41)

array-expt array pow
{gauche.array} Raises array to the power of pow; array must be a square matrix, and pow must be a nonnegative exact integer.

array-div-left a b
array-div-right a b
{gauche.array} Inverse of array-mul; array-div-left returns a matrix M such that (array-mul B M) equals to A, and array-div-right returns a matrix M such that (array-mul M B) equals to A. A and B must be a 2-dimensional square matrix. If B isn’t regular, an error is thrown.

array-add-elements array array-or-scalar ...
array-add-elements! array array-or-scalar ...
array-sub-elements array array-or-scalar ...
array-sub-elements! array array-or-scalar ...
array-mul-elements array array-or-scalar ...
array-mul-elements! array array-or-scalar ...
array-div-elements array array-or-scalar ...
array-div-elements! array array-or-scalar ...
{gauche.array} Element-wise arithmetics. The second argument and after must be an array of the same shape of the first argument, or a number; if it is a number, it is interpreted as an array of the same shape of the first argument, and each element of which is the given number.

Returns an array of the same shape of the first argument, where each element is the result of addition, subtraction, multiplication or division of the corresponding elements of the arguments.

The linear-update version (procedures whose name ends with !) may reuse the storage of the first array to calculate the result. The first array must be mutable. The caller must still use the returned value instead of counting on the side effects.
(array-add-elements (array (shape 0 2 0 2) 1 2 3 4) 10)
⇒ #,(<array> (0 2 0 2) 16 18 20 22)

(array-div-elements (array (shape 0 2 0 2) 1 3 5 7) 100)
⇒ #,(<array> (0 2 0 2) 1/200 3/400 1/120 7/800)

If only one argument is passed, these procedures returns the argument itself.
You can mix different types of arrays as long as their shapes are the same. The result is the
same type as the first argument.

(array-mul-elements (make-u8array (shape 0 2 0 2) 3) (array (shape 0 2 0 2) 1 3 5 7))
⇒ #,(<u8array> (0 2 0 2) 3 9 15 21)

$array-negate-elements array$ [Function]
$array-negate-elements! array$ [Function]

Returns an array with the same type of the shape of $array$, but each element is a negation of the

(array-negate-elements (array (shape 0 2 0 2) 1 2 3 4))
⇒ #,(<array> (0 2 0 2) -1 -2 -3 -4)

$array-reciprocate-elements array$ [Function]
$array-reciprocate-elements! array$ [Function]

Returns an array with the same type of the shape of $array$, but each element is a reciprocal of the

(array-reciprocate-elements (array (shape 0 2 0 2) 1 2 3 4))
⇒ #,(<array> (0 2 0 2) 1 1/2 1/3 1/4)

9.2 gauche.base - Importing gauche built-ins

gaufe.base [Module]

This module exports Gauche built-in procedures and syntaxes, so that they can be imported to other modules that don’t inherit gauche module.

All the bindings available in the gauche module are exported, except import, which is renamed to gauche:import to avoid conflict with R7RS import.

The module extends gauche.keyword, so also exports all the keywords—the bindings from gauche.keyword—so that the code imports gauche.base can access to self-bound keywords without inheriting the keyword module.

Typical Gauche code doesn’t need this module, for built-ins are available by default through inheritance. A newly created module inherits the gauche module by default. (See Section 4.13.5 [Module inheritance], page 83, for the details.)

Sometimes you need a module that doesn’t inherit the gauche module, yet you want to use Gauche built-in features. Particularly, R7RS libraries and programs require any bindings to be explicitly imported, so R7RS’s import and define-library sets up the module not to inherit the gauche module. In R7RS code, you need (import (gauche base)) to use Gauche’s built-in features.
Another use case is to eliminate some built-in bindings, yet keep the rest of bindings accessible, in your module. For example, the following setup creates almost-gauche module that has almost all default bindings except string-scan and string-split:

```
(define-module almost-gauche
  (use scheme.r5rs)
  (use gauche.base :except (string-scan string-split)
    :rename ((gauche:import import)))
  (extend)
)
```

```
(select-module almost-gauche)
;; your code here
```

Note the empty `extend`; it empties the module’s inheritance. (The :rename option of `gauche.base` is just to get the original name of `import` back in almost-gauche module; if you don’t use `import` directly, you won’t need it.)

## 9.3 gauche.bitvector - Bitvector utilities

This module adds more operations on bitvectors. This module and built-in bitvector procedures covers SRFI-178 Bitvector library interface (see Section 11.38 [Bitvector library], page 756)

### gauche.bitvector

Gauche supports bitvectors as a native type, but the core library only offers a handful of primitives (see Section 6.13.3 [Bitvectors], page 203). This module provides the comprehensive operations on bitvectors.

### Constructors

- `bitvector-unfold f length seed ...` [Function]
- `bitvector-unfold-right f length seed ...` [Function]

    **SRFI-178**\{gauche.bitvector\} Creates a bitvector of length `length`, and fill its contents by repeatedly applying `f`. The procedure `f` must take one more than the number of seeds, and must return the same number of values. First, `f` is applied to the index and `seed`...; then its first return value (which must be a bit, i.e. 0, 1 or a boolean value) is used to initialize the index-th element of the bitvector, and the rest of the return value is used as the next argument to `f`. It is repeated `length` times.

While `bitvector-unfold` fills the bits from left to right, `bitvector-unfold-right` does from right to left.

```
(use math.prime)
(bitvector-unfold (^
  [index n]
    (values (small-prime? n) (+ n 1)))
  20 0)
⇒ #*00110101000101000101
```

```
(bitvector-unfold-right (^
  [index n]
    (values (small-prime? n) (+ n 1)))
  20 0)
⇒ #*10100010100010101100
```

Note: This procedure follows the protocol of `vector-unfold` (see Section 10.3.2 [R7RS vectors], page 597), and has a different protocol from other `unfold` procedures (see Section 10.3.1 [R7RS lists], page 593).
bitvector-reverse-copy \textit{bv :optional start end}

[Function] \SRFI-178 \gauche.bitvector
Copies a bitvector \textit{bv} in reverse order. Optional \textit{start} and \textit{end} arguments limits the range of the input.

\begin{verbatim}
(bitvector-reverse-copy #*10110011100011110000 4 16)
⇒ #*11100011100
\end{verbatim}

bitvector-append \textit{bv} . . .

[Function] \SRFI-178 \gauche.bitvector
All arguments must be bytevectors. Returns a fresh bytevector which is concatenation of all arguments.

bitvector-concatenate \textit{bvs}

[Function] \SRFI-178 \gauche.bitvector
The argument must be a list of bitvectors. Returns a fresh bytevector which is concatenation of all bitvectors.

bitvector-append-subbitvectors \textit{bv start end} . . .

[Function] \SRFI-178 \gauche.bitvector
The number of arguments must be a multiple of 3. Each triplet of the arguments is a bytevector followed by start and end index, specifying the input bytevector range. Returns a fresh bytevector which is concatenation of all the subvectors.

\begin{verbatim}
(bitvector-append-subbitvectors
 #*11100011 2 6
 #*01010101 3 8)
⇒ #*10010101
\end{verbatim}

Predicates

bitvector-empty? \textit{bv}

[Function] \SRFI-178 \gauche.bitvector
Returns \#t if \textit{bv} is an empty bytevector, \#f if it is a non-empty bytevector. An error is thrown if \textit{bv} is not a bytevector.

bitvector=? \textit{bv} . . .

[Function] \SRFI-178 \gauche.bitvector
All arguments must be bitvectors. Returns \#t iff all the bitvectors have the same length and content. If there’s zero or one argument, it returns \#t.

Iteration

bitvector-take \textit{bv} \textit{n}

[Function] \SRFI-178 \gauche.bitvector
Returns a new bitvector with the first/last \textit{n} bits of a bitvector \textit{bv}. An error is thrown if length of \textit{bv} is less than \textit{n}.

In Gauche, you can also use generic \texttt{(subseq bv 0 n)} to take the first \textit{n} bits. See Section 9.31 [Sequence framework], page 510.

bitvector-drop \textit{bv} \textit{n}

[Function] \SRFI-178 \gauche.bitvector
Returns a new bitvector with bits except first/last \textit{n} bits of a bitvector \textit{bv}. An error is thrown if length of \textit{bv} is less than \textit{n}.

In Gauche, you can also use generic \texttt{(subseq bv n)} to drop the first \textit{n} bits. See Section 9.31 [Sequence framework], page 510.

bitvector-segment \textit{bv} \textit{n}

[Function] \SRFI-178 \gauche.bitvector
Slices a bitvector \textit{bv} with \textit{n}-bits each, and returns a list of bitvectors. If the length of \textit{bv} isn’t a multiple of \textit{n}, the last bitvector may be shorter.
Chapter 9: Library modules - Gauche extensions

**bitvector-fold/int** $kons$ $knil$ $bv1$ $bv2$ ...

**bitvector-fold/bool** $kons$ $knil$ $bv1$ $bv2$ ...

**bitvector-fold-right/int** $kons$ $knil$ $bv1$ $bv2$ ...

**bitvector-fold-right/bool** $kons$ $knil$ $bv1$ $bv2$ ...

[SRFI-178]{gauche.bitvector} Fold $kons$ over the elements of bitvectors with $knil$ as the initial state value.

The $kons$ procedure is always called as $(kons$ state $e1$ $e2$ ...), where $e1$ $e2$ ... are elements from each bitvectors, as integer 0/1 (for /int procedures) or boolean #f/#t (for /bool procedures). This is the same order as vector-fold in scheme.vector module (see Section 10.3.2 [R7RS vectors], page 597). (Note that list fold procedure calls the $kons$ procedure with different order; see Section 6.6.6 [Walking over lists], page 147).

All bitvectors must have the same length; otherwise, an error is thrown.

**bitvector-map/int** $f$ $bv1$ $bv2$ ...

**bitvector-map/bool** $f$ $bv1$ $bv2$ ...

[SRFI-178]{gauche.bitvector} Apply a procedure $f$ over each element taken from given bitvectors. The result of the procedure is gathered as a fresh bitvector and returned.

\[(\text{bitvector-map/int } \text{logand } \#*10101 \#*00111) \Rightarrow \#*00101\]

All bitvectors must have the same length; otherwise, an error is thrown.

**bitvector-map!/int** $f$ $bv1$ $bv2$ ...

**bitvector-map!/bool** $f$ $bv1$ $bv2$ ...

[SRFI-178]{gauche.bitvector} Apply a procedure $f$ over each element taken from given bitvectors. The result of the procedure is set into $bv1$. These procedure returns undefined value; you should count on the side effect onto $bv1$. In other words, they are not linear-updating procedures.

All bitvectors must have the same length; otherwise, an error is thrown.

**bitvector-map->list/int** $f$ $bv1$ $bv2$ ...

**bitvector-map->list/bool** $f$ $bv1$ $bv2$ ...

[SRFI-178]{gauche.bitvector} Apply a procedure $f$ over each element taken from given bitvectors. The result of the procedure is gathered as a list and returned.

All bitvectors must have the same length; otherwise, an error is thrown.

**bitvector-value-map-index->list** $f$ $bv$ $val$

{gauche.bitvector} Apply a procedure $f$ over each index such that the index-th element of $bv$ is $val$, which must be either a boolean value or 0 or 1. The result of applications are gathered into a list, from smaller index.

For example, the following returns a list of indices where the bit is true.

\[(\text{bitvector-value-map-index->list } \text{identity } \#*10011 \#t) \Rightarrow (0 \ 3 \ 4)\]

**bitvector-for-each/int** $f$ $bv1$ $bv2$ ...

**bitvector-for-each/bool** $f$ $bv1$ $bv2$ ...

[SRFI-178]{gauche.bitvector} Apply a procedure $f$ over each element taken from given bitvectors. The result of the procedure is discarded.

All bitvectors must have the same length; otherwise, an error is thrown.

\[(\text{bitvector-for-each/int } \text{print } \#*11100 \#*01110 \#00111) \Rightarrow \text{prints 100}\]
bitvector-value-for-each-index $f$ $bv$ $val$  
{gauche.bitvector} Apply a procedure $f$ over each index such that the index-th element of $bv$ is $val$, which must be either a boolean value or 0 or 1. The result of $f$ is discarded.

```scheme
(bitvector-value-for-each-index print #*1101111010 #f)
⇒ prints 2 7 9
```

See also bitvector-value-map-index->list above.

Prefixes, suffixes, trimming and padding

bitvector-prefix-length $bv_1$ $bv_2$  
bitvector-suffix-length $bv_1$ $bv_2$  
{SRFI-178}{gauche.bitvector} Returns the length of common prefix/suffix of two bitvectors $bv_1$ and $bv_2$.

```scheme
(bitvector-prefix-length #*1100101001 #*110001101) ⇒ 4
(bitvector-suffix-length #*1100101001 #*110001101) ⇒ 2
```

bitvector-prefix? $needle$ $haystack$  
bitvector-suffix? $needle$ $haystack$  
{SRFI-178}{gauche.bitvector} Both arguments must be bitvectors. Returns #t iff $needle$ is a prefix/suffix of $haystack$.

```scheme
(bitvector-prefix? #*101 #*1010100) ⇒ #t
(bitvector-prefix? #*101 #*1000110) ⇒ #f
(bitvector-suffix? #*110 #*1000110) ⇒ #t
(bitvector-suffix? #*110 #*1010100) ⇒ #f
```

bitvector-pad $bit$ $bv$ $len$  
bitvector-pad-right $bit$ $bv$ $len$  
{SRFI-178}{gauche.bitvector} If the length of a bitvector $bv$ is smaller than $len$, returns a bitvector of length $len$ by adding $bit$ before/after $bv$. If the length of $bv$ is greater than $len$, returns a bitvector of length $len$ such that $bv$ left/right bits of $bv$ are trimmed. If the length of $bv$ is equal to $len$, a copy of $bv$ is returned as is.

```scheme
(bitvector-pad 1 #*000101000) ⇒ #*1111100010
(bitvector-pad-right 1 #*000101000) ⇒ #*00010110110

(bitvector-pad 0 #*0100011 3) ⇒ #*011
(bitvector-pad-right 0 #*0100011 3) ⇒ #*010
```

bitvector-trim $bit$ $bv$  
bitvector-trim-right $bit$ $bv$  
bitvector-trim-both $bit$ $bv$  
{SRFI-178}{gauche.bitvector} Returns a fresh bitvector without preceding and/or trailing consecutive $bit$ from $bv$.

```scheme
(bitvector-trim 0 #*000101000) ⇒ #*101000
(bitvector-trim-right 0 #*000101000) ⇒ #*000101
(bitvector-trim-both 0 #*000101000) ⇒ #*101
```
Mutators

\texttt{bitvector-swap! bv i j} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Swap the \textit{i}-th value and \textit{j}-th value of a bitvector \textit{bv}. Returns an undefined value. If the index is out of range, an error is thrown.

\texttt{bitvector-reverse! bv :optional start end} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Reverse the order of the content of a bitvector \textit{bv} destructively. The optional \textit{start/end} indexes limit the range of the operation; elements outside of the range won’t be affected. Returns an undefined value.

\texttt{bitvector-reverse-copy! to tstart from :optional start end} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Same as \texttt{(bitvector-copy! to start (bitvector-reverse-copy from start end))}, but potentially more efficient. Returns an undefined value.

Conversion

\texttt{bitvector->list/int bv :optional start end} \hspace{1cm} [Function] \\
\texttt{bitvector->list/bool bv :optional start end} \hspace{1cm} [Function] \\
\texttt{reverse-bitvector->list/int bv :optional start end} \hspace{1cm} [Function] \\
\texttt{reverse-bitvector->list/bool bv :optional start end} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Returns a list of integers (0 or 1) or boolean values of bitvector \textit{bv}. The first two procedures returns the leftmost bit first, while the reverse- variations returns the rightmost bit first. The optional \textit{start/end} arguments limit the range of \textit{bv} to be extracted.

\begin{verbatim}
(bitvector->list/int #*010011000111 2 10) \\
⇒ (0 0 1 1 0 0 0 1)
\end{verbatim}

\begin{verbatim}
(reverse-bitvector->list/int #*010011000111 2 10) \\
⇒ (1 0 0 0 1 1 0 0)
\end{verbatim}

\texttt{reverse-list->bitvector lis} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Like \texttt{list->bitvector} (see Section 6.13.3 [Bitvectors], page 203), but pack bits from right to left.

\begin{verbatim}
(reverse-list->bitvector '(1 0 #f #f #t #t)) \\
⇒ #*110001
\end{verbatim}

\texttt{bitvector->vector/int bv :optional start end} \hspace{1cm} [Function] \\
\texttt{bitvector->vector/bool bv :optional start end} \hspace{1cm} [Function] \\
\texttt{reverse-bitvector->vector/int bv :optional start end} \hspace{1cm} [Function] \\
\texttt{reverse-bitvector->vector/bool bv :optional start end} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Returns a vector of integers (0 or 1) or boolean values of bitvector \textit{bv}. The first two procedures returns the leftmost bit first, while the reverse- variations returns the rightmost bit first. The optional \textit{start/end} arguments limit the range of \textit{bv} to be extracted.

\begin{verbatim}
(bitvector->vector/int #*010011000111 2 10) \\
⇒ (0 0 1 1 0 0 0 1)
\end{verbatim}

\begin{verbatim}
(reverse-bitvector->vector/int #*010011000111 2 10) \\
⇒ (1 0 0 0 1 1 0 0)
\end{verbatim}

\texttt{vector->bitvector vec :optional start end} \hspace{1cm} [Function] \\
\texttt{reverse-vector->bitvector vec :optional start end} \hspace{1cm} [Function] \\
\texttt{SRFI-178\{gauche.bitvector\}} Converts \textit{vec} to a fresh bitvector; \textit{vec} must be a vector of 0/1 or boolean values. The first procedure packs bits in the same left to right order, while the reverse- version reverses the order. The optional \textit{start/end} arguments limit the range of \textit{vec} to be extracted.

\begin{verbatim}
(vector->bitvector #(0 0 1 1 #t #f #t #t) 2 7) \\
⇒ #*110001
\end{verbatim}
(reverse-vector->bitvector #(0 0 1 1 #t #f #t #t) 2 7)  
⇒ #*10111

(bitvector->integer bv)  
(integer->bitvector n :optional len)

[SRFI-178]{gauche.bitvector} Convert between bitvectors and exact nonnegative integers.  
Note that the leftmost bit of bv corresponds to the least significant bit.

(bitvector->integer #*1000)  
⇒ 1
(bitvector->integer #*0001)  
⇒ 8

This is consistent with bits->list etc. of SRFI-151 (see Section 10.3.22 [R7RS bitwise  
operations], page 664), where least significant bit corresponds to the first element of the list.  
integer->bitvector takes an optional argument len to specify the number of bits in n (count  
from the least significant bit) to be taken. When omitted, (integer-length n) is used (see  
Section 6.3.6 [Basic bitwise operations], page 136).

(integer->bitvector 11)  
⇒ #*1101
(integer->bitvector 11 8)  
⇒ #*11010000

Generators

(bitvector->index-generator bv val :optional start end)

{gauche.bitvector} Returns a generator that generates indexes of bv such that the index-
th value of bv matches val, which must be either 0, 1 or a boolean value. The bitvector is  
scanned from left to right. The optional start/end arguments limits the range of the bitvector  
to be used.

(generator->list (bitvector->index-generator #*101110001 1))  
⇒ (0 2 3 4 8)

The names are consistent with Gauche convention. SRFI-178 defines these procedures with  
the name make-bitvector/int-generator and make-bitvector/bool-generator.  
See Section 9.12 [Generators], page 436, for the details of generators.

(make-bitvector-accumulator)

[SRFI-178]{gauche.bitvector} Returns an accumulator that collects given bits, and returns  
a bitvector at the end of accumulation.

(let ((a (make-bitvector-accumulator)))  
(a 1)
See Section 10.3.12 [R7RS generators], page 631, for the details of accumulators.

**Bitwise operations**

- **bitvector-not** \(bv\)  
  \([\text{Function}]\)
  Returns a bitvector whose bits are logical negation of \(bv\).

- **bitvector-not!** \(bv\)  
  \([\text{Function}]\)
  The linear updating version \(\text{bitvector-not!}\) may reuse \(bv\) to produce the result.

- **bitvector-and** \(bv1 \ bv2 \ bv \ldots\)  
  \([\text{Function}]\)
  Returns a bitvector, where each bit is the result of logical operation of corresponding bits of \(bv1\) and \(bv2\).

- **bitvector-and!** \(bv1 \ bv2 \ bv \ldots\)  
  \([\text{Function}]\)
  Returns a bitvector whose bits are the result of logical operation of corresponding bits of the given bitvectors. The linear updating version (procedures with !) may reuse \(bv1\) to produce the result.

When more than two bitvectors are given to xor and eqv, operation, if works associatively, that is,

\[
\text{bitvector-xor (bitvector-xor a b) c) = (bitvector-xor a (bitvector-xor b c))}
\]

\[
\text{bitvector-eqv (bitvector-eqv a b) c) = (bitvector-eqv a (bitvector-eqv b c))}
\]
orc1 : (or (not bv1) bv2)
orc2 : (or bv1 (not bv2))

The linear updating version (procedures with !) may reuse bv1 to produce the result.

**Quasi-integer operations**

- **bitvector-logical-shift bv count bit**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector} Returns a bitvector with the same length of bv, but the content is shifted count bits to the left (when count \(\geq 0\)) or -count bits to the right (when count < 0). The vacated bits are filled with bit.

  \[
  \text{(bitvector-logical-shift \texttt{#*101100111000} 4 \texttt{1})} \\
  \Rightarrow \texttt{#*001110001111} \\
  \text{(bitvector-logical-shift \texttt{#*101100111000} -4 0)} \\
  \Rightarrow \texttt{#*000010110011}
  \]

- **bitvector-count bit bv**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector} Returns the number of bit values in bv.

- **bitvector-count-run bit bv start**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector} Returns the number of consecutive bit values in bv, starting from index i.

- **bitvector-if bv-if bv-then bv-else**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector}

- **bitvector-first-bit bit bv**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector} Returns the index of the leftmost bit in bv. If bv doesn’t have bit, returns -1.

  \[
  \text{(bitvector-first-bit 1 \texttt{#*0010011})} \Rightarrow \texttt{2} \\
  \text{(bitvector-first-bit 0 \texttt{#*1111111})} \Rightarrow \texttt{-1}
  \]

**Bit field operations**

Those procedures operates on a partial field of a bitvector, between start-th bit (inclusive) and end-th bit (exclusive).

- **bitvector-field-any? bv start end**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector} Returns #t iff at least one bit between start and end in bv is 1.

  This is equivalent to (bitvector-any-value? bv 1 start end), using Gauche’s built-in procedure (see Section 6.13.3 [Bitvectors], page 203).

- **bitvector-field-every? bv start end**
  
  [Function]
  
  [SRFI-178]{gauche.bitvector} Returns #t iff every bit between start and end in bv is 1.

  This is equivalent to (bitvector-every-value? bv 1 start end), using Gauche’s built-in procedure (see Section 6.13.3 [Bitvectors], page 203).

- **bitvector-field-clear bv start end**
  
  [Function]

- **bitvector-field-clear! bv start end**
  
  [Function]

- **bitvector-field-set bv start end**
  
  [Function]

- **bitvector-field-set! bv start end**
  
  [Function]

[SRFI-178]{gauche.bitvector} Returns a bitvector with the same content as bv, except the bits between start (inclusive) and end (exclusive) are cleared or set, respectively.
The line-updating version may mutate \( \text{bv} \) to produce the result.

\[
\text{(bitvector-field-clear #*10010010101001 2 7)} \\
\Rightarrow \text{#*10000000101001}
\]

\[
\text{(bitvector-field-set #*10010010101001 2 7)} \\
\Rightarrow \text{#*10111110101001}
\]

\[
\text{bitvector-field-replace} \quad \text{to from start end} \\
\text{bitvector-field-replace!} \quad \text{to from start end}
\]

[SRFI-178]{gauche.bitvector} Returns a bitvector with the same content as \( \text{to-bv} \), except its bits between \( \text{start} \) (inclusive) and \( \text{end} \) (exclusive) are replaced with the content of \( \text{from-bv} \).

The line-updating version may mutate \( \text{to-bv} \) to produce the result.

\[
\text{(bitvector-field-replace #*10010010101001 #*110011 2 8)} \\
\Rightarrow \text{#*10110011101001}
\]

The length of \( \text{from-bv} \) must be at least \( \text{end} - \text{start} \). If it is longer, the exceeding bits are ignored.

See also \text{bitvector-field-replace\text{-}same} below.

\[
\text{bitvector-field-replace\text{-}same} \quad \text{to-bv from-bv start end} \\
\text{bitvector-field-replace\text{-}same!} \quad \text{to-bv from-bv start end}
\]

[SRFI-178]{gauche.bitvector} Returns a bitvector with the same content as \( \text{to-bv} \), except its bits between \( \text{start} \) (inclusive) and \( \text{end} \) (exclusive) are replaced with the content of \( \text{from-bv} \) between \( \text{start} \) and \( \text{end} \).

The line-updating version may mutate \( \text{to-bv} \) to produce the result.

\[
\text{(bitvector-field-replace\text{-}same #*10010010101001 #*11001100 2 8)} \\
\Rightarrow \text{#*10001100101001}
\]

The length of \( \text{from-bv} \) must be greater than or equal to \( \text{end} \).

See also \text{bitvector-field-replace} above.

\[
\text{bitvector-field-rotate} \quad \text{bv count start end} \\
\text{bitvector-field-flip} \quad \text{bv start end} \\
\text{bitvector-field-flip!} \quad \text{bv start end}
\]

[SRFI-178]{gauche.bitvector} Returns a bitvector whose content is the same as \( \text{bv} \) except the range between \( \text{start} \)-th bit (inclusive) and \( \text{end} \)-th bit (exclusive) being rotated by \( \text{count} \) bits toward right. You can give a negative \( \text{count} \) to rotate towards left.

\[
\text{(bitvector-field-rotate #*001100110011 1 4 8)} \\
\Rightarrow \text{#*001101100011}
\]

\[
\text{(bitvector-field-flip #*001100110011 4 8)} \\
\Rightarrow \text{#*001111000111}
\]

9.4 \text{gauche.cgen} - Generating C code

Significant part of Gauche is written in Gauche or S-expression based DSL. During the building process, they are converted into C sources and then compiled by C compiler. The \text{gauche.cgen} module and its submodules expose the functionality Gauche build process is using to the general use.
Required features for a C code generator differ greatly among applications, and too much scaffolding could be a constraint for the module users. So, instead of providing a single solid framework, we provide a set of loosely coupled modules so that you can combine necessary features freely. In fact, some of Gauche build process only use `gauche.cgen.unit` and `gauche.cgen.literal` (see `src/builtin-syms.scm`, for example).

```
module gauche.cgen

This is a convenience module that extends `gauche.cgen.unit`, `gauche.cgen.literal`, `gauche.cgen.type` and `gauche.cgen.cise` together.

Usually you can just use `gauche.cgen` and don’t need to think about individual submodules. The following subsections are organized by submodules only for the convenience of explanation.

## 9.4.1 Generating C source files

One of the tricky issues about generating C source is that you have to put several fragments of code in different parts of the source file, even you want to say just one thing—that is, sometimes you have to put declaration before the actual definition, plus some setup code that needs to be run at initialization time. The `<cgen-unit>` class takes care of such code placement.

### Creating a frame

```
<abstract> <cgen-unit>

{gauche.cgen} A `cgen-unit` is a unit of C source generation. It corresponds to one `.c` file, and optionally one `.h` file. During the processing, a "current unit" is kept in a parameter `cgen-current-unit`, and most cgen APIs implicitly work to it.

The following slot are for public use. They are used to tailor the output. Usually you set those slots at initialization time. The effect is undefined if you change them in the middle of the code generation process.

- **name**
  - [Instance Variable of `<cgen-unit>`]
  - A string to name this unit. This is used for the default name of the generated files (`name.c` and `name.h`) and the suffix of the default name of initialization function. Other cgen modules may use this to generate names. Avoid using characters that are not valid for C identifiers.

- **c-file**
  - [Instance Variable of `<cgen-unit>`]
  - The name of the C source file, in strings. If they are #f (by default), the value of `name` slot is used as the file name, with extension `.c` or `.h` is attached, respectively.

- **h-file**
  - [Instance Variable of `<cgen-unit>`]
  - The name of the C source file and header file, in strings. If they are #f (by default), the value of `name` slot is used as the file name, with extension `.c` or `.h` is attached, respectively.

- **preamble**
  - [Instance Variable of `<cgen-unit>`]
  - A list of strings to be inserted at the top of the generated sources. The default value is `("/* Generated by gauche.cgen */")`. Each string appears in its own line.

- **init-prologue**
  - [Instance Variable of `<cgen-unit>`]
  - A string to start or to end the initialization function, respectively. The default value of `init-prologue` is "void Scm_Init_NAME(void) {"} where `NAME` is the value of the `name` slot. The default value of `init-epilogue` is just "}". Each string appears in its own line.

To get the default initialization function name, use `cgen-unit-init-name` generic function.
```
To customize initialization function name, arguments and/or return type, set init-prologue.

The content of initialization function is filled by the code fragments registered by cgen-init.

**cgen-current-unit**

A parameter to keep the current cgen-unit.

A typical flow of generating C code is as follows:

1. Create a `<cgen-unit>` instance and make it the current unit.
2. Call code insertion APIs with code fragments. Fragments are accumulated in the current unit.
3. Call `emit` method (cgen-emit-c, cgen-emit-h) on the unit, which generates a C file and optionally a header file.

**cgen-emit-c cgen-unit**

**cgen-emit-h cgen-unit**

{gauche.cgen} Write the accumulated code fragments in `cgen-unit` to a C source file and C header file. The name of the files are determined by calling cgen-unit-c-file and cgen-unit-h-file, respectively. If the files already exist, its content is overwritten; you can't gradually write to the files. So, usually these procedures are called at the last step of the code generation.

We’ll explain the details of how each file is organized under “Filling the content” section below.

**cgen-unit-c-file cgen-unit**

**cgen-unit-h-file cgen-unit**

{gauche.cgen} Returns a string that names C source and header file for `cgen-unit`, respectively. The default method first looks at `c-file` or `h-file` slot of the `cgen-unit`, and if it is #f, use the value of name slot and appends an extension .c or .h.

**cgen-unit-init-name cgen-unit**

{gauche.cgen} Returns a string that names the initialization function generated to C. It is used to create the default init-prologue value.

**Filling the content**

There are four parts to which you can add C code fragment. Within each part, code fragments are rendered in the same order as added.

**extern** This part is put into the header file, if exists.

**decl** Placed at the beginning of the C source, after the standard prologue.

**body** Placed in the C source, following the ‘decl’ part.

**init** Placed inside the initialization function, which appears at the end of the C source.

The following procedures are the simple way to put a source code fragments in an appropriate part:

**cgen-extern code ...**

**cgen-decl code ...**

**cgen-body code ...**

**cgen-init code ...**

{gauche.cgen} Put code fragments code ... to the appropriate parts. Each fragment must be a string.
This is a minimal example to show the typical usage. After running this code you'll get my-cfile.c and my-cfile.h in the current directory.

```
(use gauche.cgen)

(define *unit* (make <cgen-unit> :name "my-cfile"))

(parameterize ([cgen-current-unit *unit*])
  (cgen-decl "#include <stdio.h>" )
  (cgen-init "printf(stderr, "initialization function\n");")
  (cgen-body "void foo(int n) { printf(stderr, "got %d\n", n); }")
  (cgen-extern "void foo(int n);")
)

(cgen-emit-c *unit*)
(cgen-emit-h *unit*)
```

These are handy escaping procedures; they are useful even if you don’t use other parts of the cgen modules.

- `cgen-safe-name string` [Function]
- `cgen-safe-name-friendly string` [Function]
- `cgen-safe-string string` [Function]
- `cgen-safe-comment string` [Function]

{gauche.cgen} Escapes characters invalid in C identifiers, C string literals or C comments. With `cgen-safe-name`, characters other than ASCII alphabets and digits are converted to a form _XX, where XX is hexadecimal notation of the character code. (Note that the character _ is also converted.) So the returned string can be used safely as a C identifier. The mapping is injective, that is, if the source strings differ, the result string always differ.

On the other hand, `cgen-safe-name-friendly` converts the input string into more readable C identifier. 

- `cgen-safe-string` Escapes control characters and non-ASCII characters. If the Scheme string contains a character beyond ASCII, it is encoded in Gauche’s native encoding. (NB: It also escapes ?, to avoid accidental formation of C trigraphs).

Much simpler is `cgen-safe-comment`, which just converts /* and */ into / * and * / (a space between those two characters), so that it won’t terminate the comment inadvertently. (Technically, escaping only */ suffice, but some simple-minded C parser might be confused by */ in the comments). The conversion isn’t injective as well.

```
(cgen-safe-name "char-alphabetic?") => "char_2dalphabetic_3f"
(cgen-safe-name-friendly "char-alphabetic?") => "char_alphabeticP"
(cgen-safe-string "char-alphabetic?") => "\"char-alphabetic\077\""

(cgen-safe-comment "*/*") => "*/""
```
If you want to conditionalize a fragment by C preprocessor `#ifdefs`, use the following macro:

```latex
\textbf{cgen-with-cpp-condition} \textit{cpp-expr} \textit{body} \ldots
```

{\texttt{gauche.cgen}} Code fragments submitted in \textit{body} ... are protected by \texttt{#if} \textit{cpp-expr} and \texttt{#endif}.

If \textit{cpp-expr} is a string, it is emitted literally:

\begin{verbatim}
(cgen-with-cpp-condition "defined(FOO)"
 (cgen-init "foo();"))
\end{verbatim}

;; will generate:
\begin{verbatim}
#if defined(FOO)
 foo();
#endif /* defined(FOO) */
\end{verbatim}

You can also construct \textit{cpp-expr} by S-expr.

```
<cpp-expr> : <string>
  | (defined <cpp-expr>)
  | (not <cpp-expr>)
  | (<n-ary-op> <cpp-expr> <cpp-expr> ...)
  | (<binary-op> <cpp-expr> <cpp-expr>)

<n-ary-op> : and | or | + | * | - | /

<binary-op> : > | >= | == | < | <= | !=
  | logand | logior | lognot | >> | <<
```

Example:

```
(cgen-with-cpp-condition '(and (defined FOO)
  (defined BAR))
 (cgen-init "foo();"))
```

;; will generate:
\begin{verbatim}
#if ((defined FOO)&&(defined BAR))
 foo();
#endif /* ((defined FOO)&&(defined BAR)) */
\end{verbatim}

You can nest \texttt{cgen-with-cpp-condition}.

\section*{Submitting code fragments for more than one parts}

When you try to abstract code generation process, calling individual procedures for each parts (e.g. \texttt{cgen-body} or \texttt{cgen-init}) becomes tedious, since such higher-level constructs are likely to require generating code fragments to various parts. Instead, you can create a customized class that handles submission of fragments to appropriate parts.

\begin{verbatim}
<cg-n-node>
```
{gauche.cgen} A base class to represent a set of code fragments.

The state of C preprocessor condition (set by \texttt{with-cgen-cpp-condition}) is captured when an instance of the subclass of this class is created, so generating appropriate \texttt{#ifs} and \texttt{#endifs} are automatically handled.

You subclass \texttt{<cg-n-node>}, then define method(s) to one or more of the following generic functions:
```
These generic functions are called during writing out the C source within \texttt{cgen-emit-c} and \texttt{cgen-emit-h}. Inside these methods, anything written out to the current output port goes into the output file.

While generating \texttt{.h} file by \texttt{cgen-emit-h}, \texttt{cgen-emit-xtrn} method for all submitted nodes are called in order of submission.

While generating \texttt{.c} file by \texttt{cgen-emit-c}, \texttt{cgen-emit-decl} method for all submitted nodes are called first, then \texttt{cgen-emit-body} method, then \texttt{cgen-emit-init} method.

If you don’t specialize any one of these method, it doesn’t generate code in that part.

Once you define your subclass and create an instance, you can submit it to the current cgen unit by this procedure:

\texttt{cgen-add! cgen-node} \hfill \textbf{[Function]}
\begin{verbatim}
{gauche.cgen} Submit \texttt{cgen-node} to the current cgen unit. If the current unit is not set, \texttt{cgen-node} is simply ignored.
\end{verbatim}

In fact, the procedures \texttt{cgen-extern}, \texttt{cgen-decl}, \texttt{cgen-body} and \texttt{cgen-init} are just a convenience wrapper to create an internal subclass specialized to generate code fragment only to the designated part.

### 9.4.2 Generating Scheme literals

Sometimes you want to refer to a Scheme constant value in C code. It is trivial if the value is a simple thing like Scheme boolean (\texttt{SCM_TRUE}, \texttt{SCM_FALSE}), characters (\texttt{SCM_MAKE_CHAR(code)}), small integers (\texttt{SCM_MAKE_INT(value)}), etc. You can directly write it in C code. However, once you step outside of these simple values, it gets tedious quickly, involving static data declarations and/or runtime initialization code.

For example, to get a Scheme value of a list of symbols (a b c), you have to (1) create \texttt{ScmStrings} for the names of the symbols, (2) pass them to \texttt{Scm_Intern} to get Scheme symbols, then (3) call \texttt{Scm_Conses} (or a convenience macro \texttt{SCM_LIST3}) to build a list.

With \texttt{gauche.cgen}, those code can be generated automatically.

\textbf{NOTE}: If you use \texttt{cgen-literal}, make sure you call (\texttt{cgen-decl "#include \textless gauche.h\textgreater"}) to include \texttt{gauche.h} before the first call of \texttt{cgen-literal}, which may insert declarations that needs \texttt{gauche.h}.

\texttt{cgen-literal obj} \hfill \textbf{[Function]}
\begin{verbatim}
{gauche.cgen} Returns an \texttt{<cgen-literal>} object for a Scheme object \texttt{obj}, and submit necessary declarations and initialization code to the current cgen unit.

The detail of the \texttt{<cgen-literal>} object isn’t for public use, but one thing you can do is to pass it to \texttt{cgen-cexpr} to obtain a C code fragment that refers to the Scheme value at runtime. See the example in \texttt{cgen-expr} entry below.
\end{verbatim}

\texttt{cgen-cexpr cgen-literal} \hfill \textbf{[Generic Function]}
\begin{verbatim}
{gauche.cgen} Returns a C code expression fragment of type \texttt{ScmObj}, which represents the Scheme literal value.

If the Scheme object is a immediate one such as boolean or fixnum, the C code is a immediate code to return the value, e.g. \texttt{SCM_FALSE} or \texttt{SCM_MAKE_INT(1234)}. If the Scheme object requires allocation, \texttt{cgen-literal} allocates memory and initializes it, and \texttt{cgen-cexpr} returns a pointer to that object.
\end{verbatim}
The following example creates a C function `printabc` that prints the literal value `(a b c)`, created by `cgen-literal`.

```scheme
(define *unit* (make <cgen-unit> :name "foo"))
(parameterize ((cgen-current-unit *unit*))
  (let1 lit (cgen-literal '(a b c))
    (cgen-body
      (format "void printabc() { Scm_Printf(SCM_CUROUT, "%S", ~a); }" (cgen-c-name lit))))
  (cgen-emit-c *unit*))
```

If you examine the generated file `foo.c`, you’ll get a general idea of how it is handled.

One advantage of `cgen-literal` is that it tries to share the same literal whenever possible. If you call `(cgen-literal '(a b c))` twice in the same cgen unit, you’ll get one instance of `cgen-literal`. If you call `(cgen-literal '(b c))` then, it will share the tail of the original list `(a b c)`. So you can just use `cgen-literal` whenever you need to have Scheme literal values, without worrying about generating excessive amount of duplicated code.

(Note that the literals registered with `cgen-literal` must be treated as immutable, just as in the Scheme world.)

Certain Scheme objects cannot be generated as a literal; for example, an opened port can’t, since it carries lots of runtime information.

(There’s a machinery to allow programmers to extend the `cgen-literal` behavior for new types. The API isn’t fixed yet, though.)

### 9.4.3 Conversions between Scheme and C

In the C world, Scheme objects are uniformly represented as an opaque tagged pointer `ScmObj`. In order to access the actual objects, you need to check its runtime type information and to retrieve the actual C type out of it.

**Stub types** are the objects that bridge Scheme runtime types and C types. Since mappings between Scheme types and C types are not one-to-one, there are more stub types than either types; for example, Scheme `<integer>` type may be bridged to C `int` type by the stub type `<int>`, but it may also be bridged to C `short` type by the stub type `<short>`.

Do not confuse stub types and Gauche’s runtime types—stub types are meta information associated to runtime types. You can look up a stub type by its name by `cgen-type-from-name`. The session below shows the difference of the runtime types and stub types:

```scheme
gosh> <int>
#<native-type <int>>
gosh> ,d
#<native-type <int>> is an instance of class <native-type>
slots:
  name : <int>
  super : #<class <integer>>
  c-type-name: "int"
  size : 4
  alignment : 4

gosh> (cgen-type-from-name '<int>)
#<cten-type <int>>
gosh> ,d
#<cten-type <int>> is an instance of class <cgen-type>
slots:
```
Each stub type has a C-predicate, a boxer and an unboxer, each of them is a Scheme string for the name of a C function or C macro. A C-predicate takes ScmObj object and returns C boolean value that if the given object has a valid type and range for the stub type. A boxer takes C object and converts it to a Scheme object; it usually involves wrapping or boxing the C value in a tagged pointer or object, hence the name. An unboxer does the opposite: takes a Scheme
object and convert it to a C value. The Scheme object must be checked by the C-predicate before being passed to the unboxer.

We have a few categories of stub types.

- Stub types corresponds to native types (see Section 6.1.5 [Native types], page 109).
- Stub types corresponds to C-class types. These are Scheme object whose structure is defined in C. They can be treated as ScmObj or can be casted to the specific C type; e.g. `<symbol>` can be casted to ScmSymbol*. Its unboxer is ScmObj -> C-TYPE*, and boxer is C-TYPE* -> ScmObj.
- Pass-through types. These are Scheme object that are also handled as ScmObj in C-level. Stub types only typecheck, and its boxer and unboxer are just identity. It can be either purely-Scheme-defined objects, or an object that can take multiple representations (e.g. `<integer>` can be a fixnum or ScmBignum*, so the stub generator passes through it, and the C routine handles the internals.)
- Maybe stub types. It is noted by a question mark suffix. In stub context, we only concern maybe type that can be unboxed into a C pointer type. In addition to the objects of the original type, it maps Scheme’s #f to C’s NULL and vice versa. For example, `<port>?` maps Scheme’s `<port>` instance to C’s ScmPort*, and Scheme’s #f to C’s NULL.

<cgen-type> [Class]
{gauche.cgen} An instance of this class represents a stub type. It can be looked up by name such as `<const-cstring>` by cgen-type-from-name.

cgen-type-from-name name [Function]
{gauche.cgen} Returns an instance of <cgen-type> that has name. If the name is unknown, #f is returned.

cgen-box-expr cgen-type c-expr [Function]
cgen-unbox-expr cgen-type c-expr [Function]
cgen-pred-expr cgen-type c-expr [Function]
{gauche.cgen} c-expr is a string denotes a C expression. Returns a string of C expression that boxes, unboxes, or typechecks the c-expr according to the cgen-type.

;; suppose foo() returns char*
(cgen-box-expr
 (cgen-type-from-name '<const-cstring>)
 "foo()")
⇒ "SCM_MAKE_STR_COPYING(foo())"

9.4.4 CiSE - C in S expression

Some low-level routines in Gauche are implemented in C, but they’re written in S-expression. We call it “C in S expression”, or CiSE.

The advantage of using S-expression is its readability, obviously. Another advantage is that it allows us to write macros as S-expr to S-expr translation, just like the legacy Scheme macros. That’s a powerful feature—effectively you can extend C language to suit your needs.

The gauche.cgen.cise module provides a set of tools to convert CiSE code into C code to be passed to the C compiler. It also has some support to overcome C quirks, such as preparing forward declarations.

Currently, we don’t do rigorous check for CiSE; you can pass a CiSE expression to the translator that yields invalid C code, which will cause the C compiler to emit errors. The translator inserts line directives by default so the C compiler error message points to the location of original (CiSE) source instead of generated code; however, sometimes you need to look at the generated code to figure out what went wrong. We hope this will be improved in future.
In Gauche source code, CiSE is extensively used in precompiled Scheme files and recognized by the precompiler. However, \texttt{gauche.cgen.cise} is an independent module only relies on \texttt{gauche.cgen} basic features, so you can plug it to your own C code generating programs.

### 9.4.4.1 CiSE overview

Before diving into the details, it’s easier to grasp some basic concepts.

A \textit{CiSE fragment} is an S-expression that follows CiSE syntax (see Section 9.4.4.2 [CiSE syntax], page 391). A CiSE fragment can be translated to a C code fragment by \texttt{cise-render}. Note that some translation may not be local, e.g. it may want to emit forward declarations before other C code fragments. So, the full translation requires buffering—you process all the CiSE fragments and save output, emit forward declarations, then emit the saved C code fragments. We have a wrapper procedure, \texttt{cise-translate}, to take care of it, but for your purpose you may want to roll your own wrapper.

A \textit{CiSE macro} is a Scheme code that translates a CiSE fragment to another CiSE fragment. There are number of predefined CiSE macros. You can add your own CiSE macros by utilities such as \texttt{define-cise-stmt} and \texttt{define-cise-expr}.

A \textit{CiSE ambient} is a bundle of information that affects fragment translation. It contains CiSE macro definitions, and also it keeps track of forward declarations.

If you’re not sure how a cise fragment corresponds to C code, you can interactively try it:

\begin{verbatim}
gosh> (cise-render-to-string '(.struct foo (i::int c::(const char*)))))
"struct foo { int i; const char* c; } 

gosh> (cise-render-to-string '(define-cfn foo (x::int) (return (+ x 3)))
'toplevel))
" ScmObj foo(int x){{return ((x)+(3));}}"
\end{verbatim}

(The second argument of \texttt{cise-render-to-string} specifies the context; see Section 9.4.4.3 [CiSE procedures], page 395, for the details.)

### 9.4.4.2 CiSE syntax

In this section, we lists basic CiSE syntax. They are just data from the viewpoint of Gauche—so you can build and manipulate them like any S-expression (quasiquote comes pretty handy).

#### CiSE types

C types can be written either as a symbol (e.g. \texttt{int}) or a list (e.g. \texttt{(const char *)}). When used in definition, it is preceded by ::. The following example shows types are used in local variable definitions:

\begin{verbatim}
(let* ([a :: int 0]
       [b :: (const char*) "abc"])
  ...)
\end{verbatim}

For the convenience and readability, you can write the variable name, separating double-colon and type name concatenated. You can also concatenate point suffixes (\texttt{char*} instead of \texttt{char *} in the following example):

\begin{verbatim}
(let* ([a::int 0]
       [b::(const char*) "abc"])
  ...)
\end{verbatim}

CiSE translator first breaks up these concatenated forms, then deal with types.

At this moment, CiSE does \textit{not} check if type is valid C type. It just pass along whatever given.
There are a few special type notations for more complex types. These can appear in middle of the type; for example, you can write `(const .struct x (a::int b::double) *)` to produce `const struct x {int a; double b;} *`.

```
.array elt-type (dim . . .)  [CiSE Type]
Expands to C array type, whose element type is of elt-type and dimensions are dim . . .
(cise-render-to-string '(.array char (3)))
⇒ "char [3]"
(cise-render-to-string '(.array int (2 5)))
⇒ "int [2][5]"
The last element of dim can be *, corresponds to the C type without specifying the array size:
(cise-render-to-string '(.array char (*) ))
⇒ "char [3]"
(cise-render-to-string '(.array int (10 *)) )
⇒ "int [10][]"
```

Here’s an example of global C variable definition of array type:
```
(cise-render-to-string '(define-cvar params ::(.array int (PARAM_SIZE)))
toplevel)
⇒ " int params[PARAM_SIZE];"
```

```
.struct [tag] [((field-spec . . .)]  [CiSE Type]
.union [tag] [((field-spec . . .)]  [CiSE Type]
.function (arg-spec . . .) ret-type  [CiSE Type]
```

CiSE statements
```
begin stmt . . .  [CiSE Statement]
Code grouping with { and }
let* ((name [:: type] [init-expr] . . .) stmt . . .  [CiSE Statement]
Declare and optionally assign initial values to local variables.

type should be a CiSE type. If type is omitted, the default type is ScmObj. Note that array initialization is not supported yet.

if test-expr then-stmt [else-stmt]  [CiSE Statement]
when test-expr stmt . . .  [CiSE Statement]
unless test-expr stmt . . .  [CiSE Statement]
cond (cond1 stmt1 . . . ) . . . [ (else else-stmt . . .) ]  [CiSE Statement]
Conditional statements.

case expr ((vall . . .) stmt1 . . .) . . . [ (else else-stmt . . .) ]  [CiSE Statement]
case/fallthrough expr ((vall . . .) stmt1 . . .) . . . [ (else else-stmt . . .) ]
Switch-case statement. case does not fall through between 'case' blocks while case/fallthrough does.

for (start-expr test-expr update-expr) stmt . . .  [CiSE Statement]
for () stmt . . .  [CiSE Statement]
loop stmt . . .  [CiSE Statement]
while test-expr body . . .  [CiSE Statement]
Loop statements.
for-each  (lambda  (var)  stmt . . .)  expr  

[dolost  [var  expr]  stmt . . .]  expr  must  yield  a  list.  Traverse  the  list,  binding  each  element  to  var  and  executing  stmt . . . .

The  lambda  form  is  a  fake;  you  don’t  really  create  a  closure.

pair-for-each  (lambda  (var)  stmt . . .)  expr

Like  for-each,  but  var  is  bound  to  each  ‘spine’  cell  instead  of  each  element  of  the  list.

do-pairs  [var  expr]  stmt . . .

do-times  (var  expr)  stmt . . .

expr  must  yield  an  integer,  n.  Repeat  stmt . . .  by  binding  var  from  0  to  (n-1).

return  [expr]
break
continue

Return,  break  and  continue  statements.

label  name
goto  name

Label  and  goto  statements.  We  always  add  a null  statement  after  the  label  so  that  we  can  place  (label  name)  at  the  end  of  a  compound  statement.

.if  expr  stmt  [stmt]
.when  expr  stmt . . .
.unless  expr  stmt . . .
.cond  clause . . .
.define  name[(arg . . .)]  [expr]
.undef  name
.include  path

Preprocessor  directives.

expr  could  be  a  string,  a  symbol,  a  number  or  one  of  the  following  forms:

•  (defined  c)
•  (not  c)
•  (and  c)
•  (or  c)
•  (op  c . . .)  where  op  is  either  +  or  *.
•  (op  c  c . . .)  where  op  is  either  -  or  /.
•  (op  c  c)  where  op  is  either  >,  >=,  ==,  <,  <=,  !=,  logand,  logior,  lognot,  <<  or  >>.

Note  that  defining  a  macro  function  without  value

#define  foo(abc)

is  not  supported  because  it’s  ambiguous  with

#define  foo  abc()

when  written  in  CiSE  syntax.  (.define  foo  (abc))  always  generates  the  latter.
.include  could  take  a  symbol.  This  is  used  for  including  system  header  files,  e.g.  (.include  <stdint.h>).

define-cfn  name  (arg [:  type] . . .)  [ret-type  [qualifier . . .]]  stmt

Defines  a  C  function.

If  type  or  ret-type  is  omitted,  the  default  type  is  ScmObj.
Supported  qualifiers  are  :static  and  :inline,  corresponding  to  C’s  static  and  inline  keywords.  If  :static  is  specified,  forward  declaration  is  automatically  generated.
**define-cvar** name [: type] [qualifier ...] [<init-expr>]

Defines a global C variable. Supported qualifier is :static. Note that array initialization is not supported yet.

**define-ctype** name [: type]

Defines a new type using typedef

**declare-cfn** name (arg [: type] ...) [ret-type]

Declares an external C function or variable.

**.static-decls**

Produce declarations of static functions before function bodies.

**.raw-c-code** body ...

**CiSE expressions**

- **<expr> ...**
- **- expr ...**
- *** expr ...**
- **/ expr ...**
- **% expr1 expr2**

Arithmetic operations.

- **and expr ...**
- **or expr ...**
- **not expr**

Boolean operations.

- **logand expr1 expr2 ...**
- **logior expr1 expr2 ...**
- **logxor expr1 expr2 ...**
- **lognot expr**
- **<< expr1 expr2**
- **>> expr1 expr2**

Bitwise operations.

- *** expr**
- **-> expr1 expr2 ...**
- **ref expr1 expr2 ...**
- **aref expr1 expr2 ...**
- **& expr**

Dereference, reference and address operations. **ref** is C’s .. **aref** is array reference.

- **pre++ expr**
- **post++ expr**
- **pre-- expr**
- **post-- expr**

Pre/Post increment or decrement.

- **< expr1 expr2**
- **<= expr1 expr2**
- **> expr1 expr2**
- **>= expr1 expr2**
- **== expr1 expr2**
!= expr1 expr2
  Comparison.

set! lvalue1 expr1 lvalue2 expr2 . . .
= lvalue1 expr1 lvalue2 expr2 . . .
+= lvalue expr
-= lvalue expr
*= lvalue expr
/= lvalue expr
%= lvalue expr
<<= lvalue expr
 >>= lvalue expr
logand= lvalue expr
logior= lvalue expr
logxor= lvalue expr
  Assignment expressions.

cast type expr
  Type casting.

?: test-expr then-expr else-expr
  Conditional expression.

.type type
  Useful to place a type name, e.g. an argument of sizeof operator.

9.4.4.3 CiSE procedures

cise-ambient
  {gauche.cgen}

  [Parameter]
cise-default-ambient
  {gauche.cgen}

  [Function]
cise-ambient-copy ambient
  {gauche.cgen}

  [Function]
cise-ambient-decl-strings ambient
  {gauche.cgen}

  [Function]
cise-emit-source-line
  {gauche.cgen}

  [Parameter]
cise-render cise-fragment :optional port context
  {gauche.cgen}

  [Function]
cise-render-to-string cise-fragment :optional context
  {gauche.cgen}

  [Function]
cise-render-rec cise-fragment stmt/expr env
  {gauche.cgen}

  [Function]
cise-translate inp outp :key environment
  {gauche.cgen}

  [Function]
cise-register-macro! name expander :optional ambient
  {gauche.cgen}

  [Function]
Chapter 9: Library modules - Gauche extensions

```plaintext
Chapter 9: Library modules - Gauche extensions 396

[Function]
cise-lookup-macro name :optional ambient
  {gauche.cgen}

[Macro]
define-cise-stmt name [env] clause ... [:where definition ...]
  {gauche.cgen}
define-cise-exp expr name [env] clause ... [:where definition ...]
  {gauche.cgen}
define-cise-toplevel name [env] clause ... [:where definition ...]
  {gauche.cgen}

[Macro]
define-cise-macro (name form env) body . . .
  {gauche.cgen}
define-cise-macro name name2
  {gauche.cgen}

9.4.5 Stub generation

[Stub Form]
define-type NAME C-TYPE [DESC C-PREDICATE UNBOXER BOXER]
  Register a new type to be recognized. This is rather a declaration than
definition; no C code
will be generated directly by this form.

define-cproc name (args . . .) [ret-type] [flag . . .] [qualifier . . .] stmt
  Create Scheme procedure.

args specifies arguments:
  * arg ... [:rest var] : Each arg is variable name or var::type, specifies required argu-
   ment. If :rest is given, list of excessive arguments are passed to var.
  * arg ... :optional spec ... [:rest rest-var] : Optional arguments. spec is var or
    (var default). If no default is given, var receives SCM_UNBOUND— if var isn’t a type
    of ScmObj it will raise an error.
  * ARG ... :key spec ... [:allow-other-keys [:rest rest-var]] : Keyword
    arguments. spec is var or (var default). If no default is given, var receives
    SCM_UNBOUND—if var isn’t a type of ScmObj it will raise an error.
  * arg ... :optarray (var cnt max) [:rest rest-var] : A special syntax to receive op-
    tional arguments as a C array. var is a C variable of type ScmObj*. cnt is a C variable
    of type int, which receives the number of optional argument in the ScmObj array. max
    specifies the maximum number of optional arguments that can be passed in the array
    form. If more than max args are given, a list of excessive arguments are passed to the
    rest-var if it is specified

ret-type specifies the return type of function. It could be either :: typespec or ::typespec
where typespec is a valid stub type, or (type ... ) when multiple values are returned. When
omitted, the procedure is assumed to return <top>.

flag is a keyword to modify some aspects of the procedure. Supported flags are as follows:
  * :fast-flonum - indicates that the procedure accepts flonum arguments and it won’t
    retain the reference to them. The VM can pass flonums on VM registers to the procedure
    with this flag. (This improves floating-point number handling, but it’s behavior is highly
    VM-specific; ordinary stub writers shouldn’t need to care about this flag at all.)
  * :constant - indicates that this procedure returns a constant value if all args are compile-
    time constants. The compiler may replace the call to this proc with the value, if it
determines all arguments are known at the compile time. The resulting value should be
serializable to the precompiled file.
NB: Since this procedure may be called at compile time, a subr that may return a
different value for batch/cross compilation shouldn’t have this flag.
```
qualifier is a list to adds auxiliary information to the procedure. Currently the following qualifiers are officially supported.

- (setter setter-name): specify setter. setter-name should be a cproc name defined in the same stub file.
- (setter (args ...) body ...): specify setter anonymously.
- (catch (decl c-stmt ...)): when writing a stub for C++ function that may throw an exception, use this spec to ensure the exception will be caught and converted to Gauche error condition.
- (inliner insn-name): only used in Gauche core procedures that can be inlined into an VM instruction.

stmt is a cise expression. Inside the expression, a cise macro (result expr ...) can be used to assign the value(s) to return from the cproc. As a special case, if stmt is a single symbol, it names a C function to be called with the same argument (mod unboxing) as the cproc.

define-cgeneric name c-name property-clause... [Stub Form]
Defines generic function. c-name specifies a C variable name that keeps the generic function structure. One or more of the following clauses can appear in property-clause...:
- (extern): makes c-name visible from other file (i.e. do not define the structure as static).
- (fallback "fallback"): specifies the fallback function.
- (setter . setter-spec): specifies the setter.

define-cmethod name (arg ... ) body ... [Stub Form]
define-cclass scheme-name [qualifier ...] c-type-name c-class-name cpa (slot-spec ...) property ...
Generates C stub for static class definition, slot accessors and initialization. Corresponding C struct has to be defined elsewhere.

The following qualifiers are supported:
- :base generates a base class definition (inheritable from Scheme code).
- :built-in generates a built-in class definition (not inheritable from Scheme code). This is the default if neither :base nor :built-in are specified.
- :private - the class declaration and standard macro definitions are also generated (which needs to be in the separate header file if you want the C-level structure to be used from other C code. If the extension is small enough to be contained in one C file, this option is convenient.)

cpa lists ancestor classes in precedence order. They need to be C identifiers of Scheme class Scm_*Class, for the time being. Scm_TopClass is added at the end automatically.

slot-spec is defined as (slot-name [qualifier ...]) or slot-name. The following qualifiers are supported:
- :type cgen-type
- :c-name c-name specifies the C field name if the autogenerated name from slot-name is not accurate.
- :c-spec c-spec
- :getter proc-spec specifies how to create the slot getter. proc-spec could be
  - #f to omit the getter
  - #t to generate a default one with type conversion according to type
  - A string is interpreted as the C code to implement the getter
• (c c-name) specifies the C function name that implements the getter, which is implemented elsewhere.
• :setter proc-spec specifies how to create the slot setter. The syntax is the same as :getter.

The following property are supported:
• (allocator proc-spec)
• (printer proc-spec)
• (comparer proc-spec)
• (direct-supers string ...)

define-cptr scheme-name [qualifier ...] c-type c-name c-pred [Stub Form]
c-boxer c-unboxer ([flags flag ...]) ([print print-proc] [(cleanup cleanup-proc)]
Defines a new foreign pointer class based on <foreign-pointer>. It is suitable when the C structure is mostly passed around using pointers; most typically, when the foreign library allocates the structure and returns the pointer to the Scheme world. scheme-name is a Scheme variable name. This will be bound to a newly-created subclass of <foreign-pointer> to represent this C-ptr type.

c-type is the type of the C pointer we wrap.
c-name is the C variable name (of type ScmClass *). In initialization code, an instance of a class (the same one bound to scm-name in the Scheme world) will be stored in this C variable.
c-pred is a macro name to determine if a ScmObj is of this type. c-boxer is a macro name to wrap C pointer and return a ScmObj c-unboxer is a macro name to extract C pointer from a ScmObj

The only supported qualifier is :private, which will generate c-pred, c-boxer and c-unboxer definitions automatically. Otherwise those definitions must be provided elsewhere.

The two supported flags are
• :keep-identity (which is SCM_FOREIGN_POINTER_KEEP_IDENTITY in the C world) keeps a weak hash table that maps the wrapped C pointer to the wrapping ScmObj, so Scm_MakeForeignPointer (i.e. c-boxer when :private is used) returns eq? object if the same C pointer is given.
This incurs some overhead, but cleanup procedure can safely free the foreign object without worrying if there’s other ScmObj that’s pointing to the same C pointer.
Do not use this flag if the C pointer is also allocated by GC_malloc. The used hash table is only weak for its value, so the C pointer wouldn’t be GCed.
• :map-null (which is SCM_FOREIGN_POINTER_MAP_NULL in the C world) makes Scm_MakeForeignPointer (i.e. c-boxer when :private is used) return SCM_FALSE when the C pointer is NULL.

define-symbol scheme-name [c-name] [Stub Form]
Defines a Scheme symbol. No Scheme binding is created. When c-name is given, the named C variable points to the created ScmSymbol.

define-variable scheme-name initializer [Stub Form]
Defines a Scheme variable.

define-constant scheme-name initializer [Stub Form]
Defines a Scheme constant.
define-enum name

A define-constant specialized for enum values. This is useful for exporting C enums to Scheme.

define-enum-conditionally name

Abbreviation of (if "defined(name)" (define-enum name))

define-cise-stmt name clause ...

define-cise-expr name clause ...

define-cfn ...

declare-cfn ...

define-cvar ...

declare-cvar ...

define-cftype ...

.define ...

.if ...

.include ...

.undef ...

.unless ...

.when ...

Cise macro definitions (see Section 9.4.4 [C in S expression], page 390).

initcode c-code

Insert c-code literally in the initialization function

declcode stmt ...

Inserts declaration code. stmt is usually .include or other preprocessor statements but it could also be a string which is treated as C fragments.

begin form ...

Treat each form as if they are toplevel stub forms.

if test then-stmt [else-stmt]

when test stmt

Deprecated. Please use .if and .when instead.

include file

Include and evaluate another stub file.

9.5 gauche.charconv - Character Code Conversion

gauche.charconv

This module defines a set of functions that converts character encoding schemes (CES) of the given data stream.

This module is implicitly loaded when :encoding keyword argument is given to the file stream creating functions (such as open-input-file and call-with-output-file).

For the portable programs, you can use transcoded ports defined in SRFI-181 (see Section 11.40.2 [Transcoded ports], page 761).

9.5.1 Supported character encoding schemes

A CES is represented by its name as a string or a symbol. Case is ignored. There may be several aliases defined for a single encoding.
A CES name "none" is special. When Gauche's native encoding is none, Gauche just treats a string as a byte sequence, and it's up to the application to interpret the sequence in an appropriate encoding. So, conversion to and from CES "none" does nothing.


Conversions between other encodings are handled by iconv(3) by default. However, iconv(3) API lacks a feature to customize the behavior when an input character can't be encoded in the output CES. If you need to be sensitive about it, you can disable delegation to iconv(3) by the following parameter.

```
[Parameter]

external-conversion-library

The value of this parameter can be a symbol iconv or #f. The default value is iconv.
Conversion ports opened during this parameter being iconv will use iconv(3) library if the requested conversion isn't supported by Gauche's native converters. This only affect when the conversion port is opened—once it is opened, this parameter value is irrelevant.

You can check whether the specific conversion is supported on your system or not, by the following function.
```

```
[Function]

ces-conversion-supported? from-ces to-ces

{gauche.charconv} Returns #t if conversion from the character encoding scheme (CES) from-ces to to-ces is supported in this system.

Note that this procedure may return true even if system only supports partial conversion between from-ces and to-ces. In such case, actual conversion might lose information by coercing characters in from-ces which are not supported in to-ces. (For example, conversion from Unicode to EUC-JP is "supported", although Unicode has characters that are not in EUC-JP).

Also note that this procedure always returns #t if from-ces and/or to-ces is "none", for conversion to/from CES "none" always succeeds (in fact, it does nothing).

This procedure may be affected by the value of the parameter external-conversion-library.

;; see if you can convert the internal encoding to EUC-JP
(ces-conversion-supported? (gauche-character-encoding) "euc-jp")
```

Also there are two useful procedures to deal with CES names.

```
[Function]

ces-equivalent? ces-a ces-b :optional unknown-value

{gauche.charconv} Returns true if two CESes ces-a and ces-b are equivalent to the knowledge of the system. Returns false if they are not. If the system doesn’t know about equivalency, unknown-value is returned, whose default is #f.

CES "none" works like a wild card; it is "equivalent" to any CES. (Thus, ces-equivalent? is not transitive. The intended use of ces-equivalent? is to compare two given CES names and see if conversion is required or not).

(ces-equivalent? 'eucjp "EUC-JP") ⇒ #t
(ces-equivalent? 'shift_jis "EUC-JP") ⇒ #f
(ces-equivalent? "NoSuchEncoding" 'utf-8 '?) ⇒ ?
```

```
[Function]

ces-upper-compatible? ces-a ces-b :optional unknown-value

{gauche.charconv} Returns true if a string encoded in CES ces-a can also be regarded as a string encoded in ces-a without conversion, to the knowledge of the system. Returns false if not. Returns unknown-value if the system can’t determine which is the case.
```
Like ces-equivalent?, CES "none" works like a wildcard. It is upper-compatible to any CES, and any CES is upper-compatible to "none".

(ces-upper-compatible? "eucjp" "ASCII") ⇒ #t
(ces-upper-compatible? "eucjp" "utf-8") ⇒ #f
(ces-upper-compatible? "utf-8" "NoSuchEncoding" '?) ⇒ ?

When Gauche’s internal conversion routine encounters a character that can’t be mapped, the behavior depends on the illegal output handling mode of the conversion port, specified by illegal-output keyword arguments. If the mode is raise, an <io-encoding-error> is thrown. If the mode is replace, the character is replaced with a replacement character.

A replacement character is U+FFFD (REPLACEMENT CHARACTER) if it is available. For Japanese encodings, U+FFFD isn’t available, and we use U+3013 (geta mark), for it is traditionally used as the replacement character. If neither one is available, ? is used.

If that happens in iconv, handling of such character depends on iconv implementation (glibc implementation returns an error).

If the conversion routine encounters an input sequence that is illegal in the input CES, an <io-decoding-error> is signaled.

**Details of Gauche’s native conversion algorithm:** Between EUC_JP, Shift JIS and ISO2022JP, Gauche uses arithmetic conversion whenever possible. Between Unicode (UTF-8) and EUC_JP, Gauche uses lookup tables. Between Unicode and Shift JIS or ISO2022JP, Gauche converts the input CES to EUC_JP, then convert it to the output CES. ISO8859-N are converted to Unicode using tables, then converted to the output CES if necessary. If the same CES is specified for input and output, Gauche’s conversion routine just copies input characters to output characters, without checking the validity of the encodings.

**EUC_JP, EUCJP, EUCJ, EUC_JISX0213**
Covers ASCII, JIS X 0201 kana, JIS X 0212 and JIS X 0213 character sets. JIS X 0212 character set is supported merely because it uses the code region JIS X 0213 doesn’t use, and JIS X 0212 characters are not converted properly to Shift JIS and UTF-8. Use JIS X 0213.

**SHIFT_JIS, SHIFITJIS, SJIS**
Covers Shift_JISX0213, except that 0x5c and 0x7e is mapped to ASCII character set (REVERSE SOLIDUS and TILDE), instead of JIS X 0201 Roman (YEN SIGN and OVERLINE).

**UTF-8, UTF8**
Unicode. Note that some JIS X 0213 characters are mapped to Extension B (U+20000 and up). Some JIS X 0213 characters are mapped to two unicode characters (one base character plus a combining character).

These encodings differ a bit (except ISO2022JP and CSISO2022JP, which are synonyms), but Gauche handles them same. If one of these CES is specified as input, Gauche recognizes escape sequences of any of CES. ISO2022JP-2 defines several non-Japanese escape sequences, and they are recognized by Gauche, but mapped to substitution character (’?’ or geta mark).

For output, Gauche assumes ISO2022JP first, and uses ISO2022JP-1 escape sequence to put JIS X 0212 character, or uses ISO2022JP-3 escape sequence to put JIS X 0213 plane 2 character. Thus, if the string contains only JIS X 0208 characters, the output is compatible to ISO2022JP. Precisely speaking, JIS X 0213 specifies some characters in JIS X 0208 codepoint that shouldn’t be mixed with JIS X 0208 characters; Gauche output those characters as JIS X 0208 for compatibility. (This is the same policy as Emacs-Mule’s iso2022jp-3-compatible mode).
Chapter 9: Library modules - Gauche extensions

9.5.2 Autodetecting the encoding scheme

There are cases that you don’t know the CES of the input, but you know it is one of several possible encodings. The charconv module has a mechanism to guess the input encoding. There can be multiple algorithms, and each algorithm has the name (wildcard CES). Right now, there’s only one algorithm implemented:

"*JP" To guess the character encoding from Japanese text, among either ISO2022-JP(-1,2,3), EUCJP, SHIFT-JIS or UTF-8.

The wildcard CES can be used in place of CES name for some conversion functions.

ces-guess-from-string string scheme {gauche.charconv} Guesses the CES of string by the character guessing scheme scheme (e.g. "*JP"). Returns CES name that can be used by other charconv functions. It may return #f if the guessing scheme finds no possible encoding in string. Note that if there may be more than one possible encoding in string, the guessing scheme returns one of them, usually in favor of the native CES.

9.5.3 Conversion ports

open-input-conversion-port source from-code :key to-code buffer-size owner? illegal-output {gauche.charconv} Takes an input port source, which feeds characters encoded in from-code, and returns another input port, from which you can read characters encoded in to-code. If to-code is omitted, the native CES is assumed.

buffer-size is used to allocate internal buffer size for conversion. The default size is about 1 kilobytes and it’s suitable for typical cases.

Handling argument specifies the behavior when the output CES doesn’t have the corresponding character of input. It can be a symbol raise to raise an <io-encoding-error> in such cases, or a symbol replace to replace the character with a replacement character appropriate in the output CES. If omitted, raise is assumed.

Note that iconv(3) library API doesn’t offer an option to choose the illegal-output handling mode. So when the conversion is delegated to iconv(3), illegal-output is ignored and the behavior follows the underlying iconv(3) implementation. If you need to make sure illegal-output is honored, you can bind the parameter external-conversion-library to #f when calling this procedure; then the conversion port won’t use iconv(3) and raises unsupported encodings error if the conversion can’t be handled entirely within Gauche.

By default, open-input-conversion-port leaves source open. If you specify true value to owner?, the function closes source after it reads EOF from the port.

If you don’t know the source’s CES, you can specify CES guessing scheme, such as "*JP", in place of from-code. The conversion port tries to guess the encoding, by prefetching the data from source up to the buffer size. It signals an error if the code guessing routine finds no appropriate CES. If the guessing routine finds ambiguous input, however, it silently assume one of possible CES’s, in favor of the native CES. Hence it is possible that the guessing is wrong if the buffer size is too small. The default size is usually enough for most text documents, but it may fail if the large text contains mostly ASCII characters and multibyte characters appear only at the very end of the document. To be sure for the worst case, you have to specify the buffer size large enough to hold entire text.

For example, the following code copies a file unknown.txt to a file eucjp.txt, converting unknown Japanese CES to EUC-JP.

(call-with-output-file "eucjp.txt"
(lambda (out)
(copy-port (open-input-conversion-port
  (open-input-file "unknown.txt")
  "*jp" ;guess code
  :to-code "eucjp"
  :owner? #t) ;close unknown.txt afterwards
  out)))

For the portable code, you can also use SRFI-181 transcoded-port (see Section 11.40.2 [Transcoded ports], page 761).

open-output-conversion-port sink to-code :key from-code buffer-size owner? illegal-output

{gauche.charconv} Creates and returns an output port that converts given characters from from-code to to-code and feed to an output port sink. If from-code is omitted, the native CES is assumed. You can’t specify a character guessing scheme (such as "*JP") to neither from-code nor to-code.

buffer-size specifies the size of internal conversion buffer. The characters put to the returned port may stay in the buffer, until the port is explicitly flushed (by flush) or the port is closed.

By default, the returned port doesn’t closes sink when itself is closed. If a keyword argument owner? is provided and true, however, it closes sink when it is closed.

The illegal-output keyword argument is the same as open-input-conversion-port.

For the portable code, you can also use SRFI-181 transcoded-port (see Section 11.40.2 [Transcoded ports], page 761).

ces-convert-to return-type source from-code :optional to-code :key illegal-output

ces-convert source from-code :optional to-code :key illegal-output

{gauche.charconv} Convert source, which is a string or an u8vector of multibyte encoding in from-code, to a string or u8vector encoded in to-code. If to-code is omitted, the native CES is assumed.

In ces-convert-to, you can specify the return type by return-type argument; it must be either a class object <string> or <u8vector>. On the other hand, ces-convert always returns a string, regardless of the type of source.

If to-code is different from the native CES and a string is returned, it can be an incomplete string. It’s for the backward compatibility—in general, we recommend to use u8vector to represent multibyte sequence in CES other than the native encoding.

from-code can be a name of character guessing scheme (e.g. "*JP").

The keyword argument illegal-output controls the behavior when input contains a character that can’t be encoded in the output. See open-input-conversion-port above for the description. By default, an <io-encoding-error> is raised, except when the conversion is delegated to iconv(3), in which case the behavior depends on the external library.

For the portable code, you can also use SRFI-181 bytevector->string and string->bytevector (see Section 11.40.2 [Transcoded ports], page 761).

call-with-input-conversion iport proc :key encoding conversion-buffer-size illegal-output

call-with-output-conversion oport proc :key encoding conversion-buffer-size illegal-output

{gauche.charconv} These procedures can be used to perform character I/O with different encoding temporary from the original port’s encoding.
call-with-input-conversion takes an input port iport which uses the character encoding encoding, and calls proc with one argument, a conversion input port. From the port, proc can read characters in Gauche’s internal encoding. Note that once proc is called, it has to read all the characters until EOF; see the note below.

call-with-output-conversion takes an output port oport which expects the character encoding encoding, and calls proc with one argument, a temporary conversion output port. To the port, proc can write characters in Gauche’s internal encoding. When proc returns, or it exits with an error, the temporary conversion output port is flushed and closed. The caller of call-with-output-conversion can continue to use oport with original encoding afterwards.

Both procedure returns the value(s) that proc returns. The default value of encoding is Gauche’s internal encoding. Those procedures don’t create a conversion port when it is not necessary. If conversion-buffer-size is given, it is used as the buffer-size argument when the conversion port is open.

You shouldn’t use iport/oport directly while proc is active—character encoding is a stateful process, and mixing I/O from/to the conversion port and the underlying port will screw up the state.

Note: for the call-with-input-conversion, you can’t use iport again unless proc reads EOF from it. It’s because a conversion port needs to buffer the input, and there’s no way to undo the buffered input to iport when proc returns.

with-input-conversion iport thunk :key encoding conversion-buffer-size [Function] illegal-output

with-output-conversion oport thunk :key encoding conversion-buffer-size illegal-output

{gauche.charconv} Similar to call-with-*-conversion, but these procedures call thunk without arguments, while the conversion port is set as the current input or output port, respectively. The meaning of keyword arguments are the same as call-with-*-conversion.

wrap-with-input-conversion port from-code :key to-code owner? buffer-size illegal-output

wrap-with-output-conversion port to-code :key from-code owner? buffer-size illegal-output

{gauche.charconv} Convenient procedures to avoid adding unnecessary conversion port. Each procedure works like open-input-conversion-port and open-output-conversion-port, respectively, except if system knows no conversion is needed, no conversion port is created and port is returned as is.

When a conversion port is created, port is always owned by the port. When you want to close the port, always close the port returned by wrap-with-*-conversion, instead the original port. If you close the original port first, the pending conversion won’t be flushed. (Some conversion requires trailing sequence that is generated only when the conversion port is closing, so simply calling flush isn’t enough.)

The buffer-size and illegal-output arguments are passed to the open-*-conversion-port.

9.6 gauche.collection - Collection framework

gauce.collection [Module]

This module provides a set of generic functions (GFs) that iterate over various collections. The Scheme standard has some iterative primitives such as map and for-each, and scheme.list (see Section 10.3.1 [R7RS lists], page 593, adds a rich set of such functions, but they work only on lists.
Using the method dispatch of the object system, this module efficiently extends those functions for other collection classes such as vectors and hash tables. It also provides a simple way for user-defined class to adapt those operations. So far, the following operations are defined.

Mapping  fold, fold2, fold3, map, map-to, map-accum, for-each
Selection and searching  find, find-min, find-max, find-minmax, filter, filter-to, remove, remove-to, partition, partition-to group-collection
Conversion  coerce-to
Miscellaneous  size-of, lazy-size-of

Those operations work on collections and its subclass, sequences. A collection is a certain form of a set of objects that you can traverse all the object in it in a certain way. A sequence is a collection that all its elements are ordered, so that you can retrieve its element by index.

The following Gauche built-in objects are treated as collections and/or sequences.

- `<list>`  A sequence.
- `<vector>`  A sequence.
- `<string>`  A sequence (of characters)
- `<hash-table>`  A collection. Each element is a pair of a key and a value.
- `<s8vector>, <u8vector>, ... <f64vector>`  A sequence (methods defined in `gauche.uvector` module, see Section 6.13.2 [Uniform vectors], page 198).

See Section 9.31 [Sequence framework], page 510, for it adds more sequence specific methods. The methods that needs to return a set of objects, i.e. map, filter, remove and partition, returns a list (or lists). The corresponding “-to” variant (map-to, filter-to, remove-to and partition-to) takes a collection class argument and returns the collection of the class.

### 9.6.1 Mapping over collection

These generic functions extends the standard mapping procedures. See also Section 9.31.3 [Mapping over sequences], page 511, if you care the index as well as elements.

**fold proc knil coll coll2 . . .**  [Generic function]

{gauche.collection} This is a natural extension of fold (see Section 6.6.7 [Other list procedures], page 151).

For each element $E_i$ in the collection $coll$, $proc$ is called as $(proc E_i R_i-1)$, where $R_i-1$ is the result of $(i-1)$-th invocation of $proc$ for $i > 0$, and $R_0$ is $knil$. Returns the last invocation of $proc$.

\[
\text{fold } + 0 \ '##(1 \ 2 \ 3 \ 4)) \Rightarrow 10 \\
\text{fold cons } '(' \ "abc") \Rightarrow (#\c \ #b \ #a)
\]

If the $coll$ is a sequence, it is guaranteed that the elements are traversed in order. Otherwise, the order of iteration is undefined.
Note: We don’t provide `fold-right` on collections, since the order of elements doesn’t matter, so only `fold` is sufficient for meaningful traversal. However, sequences do have `fold-right`: see Section 9.31.3 [Mapping over sequences], page 511.

You can fold more than one collection, although it doesn’t make much sense unless all of the collections are sequences. Suppose \( E(k, i) \) for \( i \)-th element of \( k \)-th collection. \( proc \) is called as

\[
\text{proc} \, E(0, i) \, E(1, i) \ldots \, E(K-1, i) \, R_{i-1}
\]

Different types of collections can be mixed together.

\[(\text{fold acons } '() \, "abc" \, ' #(1 2 3))\]
\[
\Rightarrow ((\#\!\!\!\!\!\!\!\! c \, 3) \, \, (\#\!\!\!\!\!\!\!\! b \, 2) \, \, (\#\!\!\!\!\!\!\!\! a \, 1))
\]

`;; calculates dot product of two vectors

\[(\text{fold } (\text{lambda } (a \, b \, r) \, (+ (* a \, b) \, r)) \, 0 \, ' #(3 \, 5 \, 7) \, ' #(2 \, 4 \, 6))\]
\[
\Rightarrow 68
\]

When more than one collection is given, `fold` terminates as soon as at least one of the collections exhausted.

\[
\text{fold2} \, \text{proc} \, \text{knil1} \, \text{knil2} \, \text{coll} \, \text{coll2} \, \ldots
\]
\[
\text{fold3} \, \text{proc} \, \text{knil1} \, \text{knil2} \, \text{knil3} \, \text{coll} \, \text{coll2} \, \ldots
\]

{gauche.collection} Like `fold`, but they can carry two and three state values instead of one, respectively. The state values are initialized by `knilN`. The procedure `proc` is called with each element of `collN`, and the state values. It must return two (`fold2`) or three (`fold3`) values, which will be used as the state values of next iteration. The values returned in the last iteration will be the return values of `fold2` and `fold3`.

\[(\text{fold2} (\text{lambda } (\text{elt} \, a \, b) \, (\text{values} \, (\text{min} \, \text{elt} \, a) \, (\text{max} \, \text{elt} \, b))) \, 256 \, 0 \, ' \#u8(33 \, 12 \, 142 \, 1 \, 74 \, 98 \, 12 \, 5 \, 99))\]
\[
\Rightarrow 1 \, \text{and} \, 142 \, ;; \text{find minimum and maximum values}
\]

See also `map-accum` below.

\[
\text{map} \, \text{proc} \, \text{coll} \, \text{coll2} \, \ldots
\]

{gauche.collection} This extends the built-in `map` (see Section 6.6.6 [Walking over lists], page 147). Apply `proc` for each element in the collection `coll`, and returns a list of the results.

If the `coll` is a sequence, it is guaranteed that the elements are traversed in order. Otherwise, the order of iteration is undefined.

If more than one collection is passed, `proc` is called with elements for each collection. In such case, `map` terminates as soon as at least one of the collection is exhausted. Note that passing more than one collection doesn’t make much sense unless all the collections are sequences.

\[(\text{map} \, (\text{lambda } (x) \, (* x \, 2)) \, ' #(1 \, 2 \, 3))\]
\[
\Rightarrow '(2 \, 4 \, 6)
\]

\[(\text{map char-upcase } "abc")\]
\[
\Rightarrow '(\#\!\!\!\!\!\!\!\! A \, \#\!\!\!\!\!\!\!\! B \, \#\!\!\!\!\!\!\!\! C)
\]

\[(\text{map } + \, ' #(1 \, 2 \, 3) \, ' #(4 \, 5 \, 6))\]
\[
\Rightarrow (5 \, 7 \, 9)
\]

`map` always returns a list. If you want to get the result in a different type of collection, use `map-to` described below. If you wonder why `(map char-upcase "abc")` doesn’t return "ABC", read the discussion in the bottom of this subsection.
**map-to** class proc coll coll2 . . .

{gauche.collection} This works the same as **map**, except the result is returned in a collection of class **class**. **Class** must be a collection class and have a builder interface (see Section 9.6.4 [Fundamental iterator creators], page 410).

\[
\text{(map-to <vector> + '1 2 3) '4 5 6)} \\
\Rightarrow (5 7 9)
\]

\[
\text{(map-to <string> char-upcase "def")} \\
\Rightarrow "DEF"
\]

\[
\text{(map-to <vector> char=? "bed" "pet")} \\
\Rightarrow (#f #t #f)
\]

**map-accum** proc seed coll1 coll2 . . .

{gauche.collection} Collects results of **proc** over collections, while passing a state value. **proc** is called like this:

\[
\text{(proc elt1 elt2 . . . seed)}
\]

Where **elt1 elt2 . . .** are the elements of **coll1 coll2 . . .**. It must return two values; the first value is collected into a list (like **map**), while the second value is passed as **seed** to the next call of **proc**.

When one of the collections is exhausted, **map-accum** returns two values, the list of the first return values from **proc**, and the second return value of the last call of **proc**.

If the given collections are sequences, it is guaranteed that **proc** is applied in order of the sequence.

This is similar to Haskell’s **mapAccumL**, but note that the order of **proc**’s argument and return values are reversed.

**for-each** proc coll coll2 . . .

{gauche.collection} Extension of built-in **for-each** (see Section 6.6.6 [Walking over lists], page 147). Applies **proc** for each elements in the collection(s). The result of **proc** is discarded. The return value of **for-each** is undefined.

If the **coll** is a sequence, it is guaranteed that the elements are traversed in order. Otherwise, the order of iteration is undefined.

If more than one collection is passed, **proc** is called with elements for each collection. In such case, **for-each** terminates as soon as one of the collection is exhausted. Note that passing more than one collection doesn’t make much sense unless all the collections are sequences.

**Discussion:** It is debatable what type of collection **map** should return when it operates on the collections other than lists. It may seem more “natural” if **(map * '#(1 2) '#(3 4))** returns a vector, and **(map char-upcase "abc")** returns a string.

Although such interface seems work for simple cases, it’ll become problematic for more general cases. What type of collection should be returned if a string and a vector are passed? Furthermore, some collection may only have iterator interface but no builder interface, so that the result can’t be coerced to the argument type (suppose you’re mapping over database records, for example). And Scheme programmers are used to think **map** returns a list, and the result of **map** are applied to the procedures that takes list everywhere.
So I decided to add another method, map-to, to specify the return type explicitly. The idea of passing the return type is taken from CommonLisp's map function, but taking a class metaobject, map-to is much flexible to extend using method dispatch. This protocol ("-to" variant takes a class metaobject for the result collection) is used throughout the collection framework.

## 9.6.2 Selection and searching in collection

### find pred coll

{gauche.collection} Applies pred for each element of a collection coll until pred returns a true value. Returns the element on which pred returned a true value, or #f if no element satisfies pred.

If coll is a sequence, it is guaranteed that pred is applied in order. Otherwise the order of application is undefined.

```scheme
(find char-upper-case? "abcDe") ⇒ #\D
(find even? ’(1 3 4 6)) ⇒ 4
(find even? '(1 3 5 7)) ⇒ #f
```

### find-min coll :key key compare default

{gauche.collection} Returns a minimum or maximum element in the collection coll.

A one-argument procedure key, whose default is identity, is applied for each element to obtain a comparison value. Then a comparison value is compared by a two-argument procedure compare, whose default is <. If the collection has zero or one element, the compare procedure is never called.

When the collection is empty, a value given to default is returned, whose default is #f.

```scheme
(find-min '((a . 3) (b . 9) (c . -1) (d . 7)) :key cdr) ⇒ (c . -1)
```

### find-min&max coll :key key compare default default-min default-max

{gauche.collection} Does find-min and find-max simultaneously, and returns two values, the minimum element and the maximum element. The keyword arguments key, compare, and default are the same as find-min and find-max. Alternatively you can give default values for minimum and maximum separately, by default-min and default-max.

### filter pred coll

{gauche.collection} Returns a list of elements of collection coll that satisfies the predicate pred. If the collection is a sequence, the order is preserved in the result.

```scheme
(filter char-upper-case? "Hello, World")
⇒ (#\H #\W)
```

### filter-to class pred coll

{gauche.collection} Same as filter, but the result is returned as a collection of class class.

```scheme
(filter-to <vector> even? ’(1 2 3 4)) ⇒ #(2 4)
```

### remove pred coll

{gauche.collection} Returns a list of elements of collection coll that does not satisfy the predicate pred. If the collection is a sequence, the order is preserved in the result.

```scheme
(remove char-upper-case? "Hello, World")
⇒ (#\e #\l #\o #\, #\space #\o #\r #\l #\d)
```
**Chapter 9: Library modules - Gauche extensions**

### 9.6.3 Miscellaneous operations on collection

#### remove-to class pred coll

{gauche.collection} Same as remove, but the result is returned as a collection of class class.

- (remove-to <vector> even? '(1 2 3 4)) ⇒ #(1 3)
- (remove-to <string> char-upper-case? "Hello, World")
  ⇒ "ello, orld"

### partition pred coll

{gauche.collection} Does filter and remove the same time. Returns two lists, the first consists of elements of the collection coll that satisfies the predicate pred, and the second consists of elements that doesn’t.

- (partition char-upper-case? "PuPu")
  ⇒ (#\P #\P) and (#\u #\u)
- (partition even? '(1 2 3 4))
  ⇒ (2 4) and (1 3)

#### partition-to class pred coll

{gauche.collection} Same as partition, except the results are returned in the collections of class class.

- (partition-to <string> char-upper-case? "PuPu")
  ⇒ "PP" and "uu"
- (partition-to <vector> even? '(1 2 3 4))
  ⇒ #(2 4) and #(1 3)

### group-collection coll :key key test

{gauche.collection} Generalized partition. Groups elements in coll into those who has the same key value, and returns the groups as of lists. Key values are calculated by applying the procedure key to each element of coll. The default value of key is identity. For each element of coll, key is applied exactly once. The equal-ness of keys are compared by test procedure, whose default is eqv?.

If coll is a sequence, then the order of elements in each group of the result is the same order in coll.

- (group-collection '(1 2 3 2 3 1 2 1 2 3 2 3))
  ⇒ ((1 1 1) (2 2 2 2 2) (3 3 3 3))
- (group-collection '(1 2 3 2 3 1 2 1 2 3 2 3) :key odd?)
  ⇒ ((1 3 1 1 3 3) (2 2 2 2 2))
- (group-collection '(({"a"} 2) ("b" 5) ("c" 1) ("b" 3) ("a" 6)) :key car :test string=?)
  ⇒ ((("a" 2) ("a" 6)) (("b" 5) ("b" 3)) (("c" 1)))

See also group-sequence in gauche.sequence (see Section 9.31.4 [Other operations over sequences], page 512), which only groups adjacent elements.

#### size-of coll

{gauche.collection} Returns the number of elements in the collection. Default method iterates over the collection to calculate the size, which is not very efficient and may diverge if the collection is infinite. Some collection classes overload the method for faster calculation.
**Chapter 9: Library modules - Gauche extensions**

**lazy-size-of** *coll*  
{gauche.collection} Returns either the size of the collection, or a promise to calculate it. The intent of this method is to avoid size calculation if it is expensive. In some cases, the caller wants to have size just for optimization, and it is not desirable to spend time to calculate the size. Such caller uses this method and just discards the information if it is a promise.

**coerce-to** *class coll*  
{gauche.collection} Convert a collection *coll* to another collection which is an instance of *class*. If *coll* is a sequence and *class* is a sequence class, the order is preserved.

(coerce-to <vector> '(1 2 3 4))
⇒ #(1 2 3 4)

(coerce-to <string> '#(#\a #\b #\c))
⇒ "abc"

### 9.6.4 Fundamental iterator creators

These are fundamental methods on which all the rest of iterative method are built. The method interface is not intended to be called from general code, but suitable for building other iterator construct. The reason why I chose this interface as fundamental methods are explained at the bottom of this subsection.

**call-with-iterator** *collection proc :key start*  
{gauche.collection} A fundamental iterator creator. This creates two procedures from *collection*, both take no argument, and then call *proc* with those two procedures. The first procedure is terminate predicate, which returns #t if the iteration is exhausted, or #f if there are still elements to be visited. The second procedure is an incrementer, which returns one element from the collection and sets the internal pointer to the next element. The behavior is undefined if you call the incrementer after the terminate predicate returns #t.

If the collection is actually a sequence, the incrementer is guaranteed to return elements in order, from 0-th element to the last element. If a keyword argument *start* is given, however, the iteration begins from *start*-th element and ends at the last element. If the collection is not a sequence, the iteration order is arbitrary, and *start* argument has no effect.

An implementation of **call-with-iterator** method may limit the extent of the iterator inside the dynamic scope of the method. For example, it allocates some resource (e.g. connect to a database) before calling *proc*, and deallocates it (e.g. disconnect from a database) after *proc* returns.

This method returns the value(s) *proc* returns.

(call-with-iterator '(1 2 3 4 5)
  (lambda (end? next)
    (do ((odd-nums 0))
      ((end?) odd-nums)
      (when (odd? (next)) (inc! odd-nums)))))
⇒ 3

See also **with-iterator** macro below, for it is easier to use.

**with-iterator** *(collection end? next args ...) body ...*  
{gauche.collection} A convenience macro to call **call-with-iterator**.

(with-iterator (coll end? next args ...) body ...)
≡
(call-with-iterator coll
(lambda (end? next) body ...)  
args ...)

call-with-iterators collections proc  
{gauche.collection} A helper function to write n-ary iterator method. This function applies call-with-iterator for each collections, and makes two lists, the first consists of terminate predicates and the second of incrementers. Then proc is called with those two lists. Returns whatever proc returns.

call-with-builder collection-class proc :key size  
{gauche.collection} A fundamental builder creator. Builder is a way to construct a collection incrementally. Not all collection classes provide this method.

Collection-class is a class of the collection to be built. This method creates two procedures, adder and getter, then calls proc with those procedures. Adder procedure takes one argument and adds it to the collection being built. Getter takes no argument and returns a built collection object. The effect is undefined if adder is called after getter is called.

A keyword argument size may be specified if the size of the result collection is known. Certain collections may be built much more efficiently if the size is known; other collections may just ignore it. The behavior is undefined if more than size elements are added, or the collection is retrieved before size elements are accumulated.

If the collection class is actually a sequence class, adder is guaranteed to add elements in order. Otherwise, the order of elements are insignificant.

Some collection class may take more keyword arguments to initialize the collection.

This method returns the value(s) proc returned.

(call-with-builder <list>  
  (lambda (add! get)  
    (add! 'a) (add! 'b) (add! 'c) (get)))  
⇒ (a b c)

(call-with-builder <vector>  
  (lambda (add! get)  
    (add! 'a) (add! 'b) (add! 'c) (get)))  
⇒ #(a b c)

See also with-builder macro below, for it is much easier to use.

with-builder (collection add! get args ...) body ...  
{gauche.collection} A convenience macro to call call-with-builder.

(with-builder (coll add! get args ...) body ...)  
≡  
(call-with-builder coll  
  (lambda (add! get) body ...)  
  args ...)

Discussion: Other iterator methods are built on top of call-with-iterator and call-with-builder. By implementing those methods, you can easily adapt your own collection class to all of those iterative operations. Optionally you can overload some of higher-level methods for efficiency.

It is debatable that which set of operations should be primitives. I chose call-with-iterator style for efficiency of the applications I see most. The following is a discussion of other possible primitive iterators.

fold  It is possible to make fold a primitive method, and build other iterator method on top of it. Collection-specific iterating states can be kept in the stack of fold,
thus it runs efficiently. The method to optimize a procedure that uses fold as a basic iterator construct. However, it is rather cumbersome to derive generator-style interface from it. It is also tricky to iterate irregularly over more than one collections.

CPS Passes iteratee the continuation procedure that continues the iteration. The iteratee just returns when it want to terminate the iteration. It has resource management problem described in Oleg Kiselyov’s article (http://okmij.org/ftp/Scheme/enumerators-callcc.html).

Iterator object
Like C++ iterator or Common Lisp generator. Easy to write loop. The problem is that every call of checking termination or getting next element must be dispatched.

Series Common Lisp’s series can be very efficient if the compiler can statically analyze the usage of series. Unfortunately it is not the case in Gauche. Even if it could, the extension mechanism doesn’t blend well with Gauche’s object system.

Macros Iterator can be implemented as macros, and that will be very efficient; e.g. Scheme48’s iterator macro. It uses macros to extend, however, and that doesn’t blend well with Gauche’s object system.

The current implementation is close to the iterator object approach, but using closures instead of iterator objects so that avoiding dispatching in the inner loop. Also it allows the iterator implementor to take care of the resource problem.

9.6.5 Implementing collections
The minimum requirements of the collection class implementation is as follow:

- The class inherits <collection> abstract class.
- A method call-with-iterator is implemented.

This makes iterator methods such as map, for-each, find and filter to work.

In order to make the constructive methods (e.g. map-to to create your collection), you have to implement call-with-builder method as well. Note that call-with-builder method must work a sort of class method, dispatched by class, rather than normal method dispatched by instance. In Gauche, you can implement it by using a metaclass. Then the minimal code will look like this:

```scheme
(define-class <your-collection-meta> (<class>) ()

(define-class <your-collection> (<collection>)
  (...); slots
  :metaclass <your-collection-meta>)

(define-method call-with-iterator
  ((coll <your-collection>) proc . options)
  ...
)

(define-method call-with-builder
  ((coll <your-collection-meta>) proc . options)
  ...
)
```

Optionally, you can overload other generic functions to optimize performance.
9.7 gauche.config - Configuration parameters

gauche.config
This module allows the Scheme program to access the configuration information the same as you can get from the gauche-config program.

gauche-config option
{gauche.config} Returns the configured value of the option.
See the manpage of gauche-config, or run gauche-config without any argument from the shell, to find out the valid options.

(gauche-config "--cc")
⇒ "gcc"
(gauche-config "-L")
⇒ "-L/usr/lib/gauche/0.6.5/i686-pc-linux-gnu"
(gauche-config "-l")
⇒ "-ldl -lcrypt -lm -lpthread"

9.8 gauche.configure - Generating build files

gauche.configure
This is a utility library to write a configure script. It is used to check the system properties and generates build files (usually Makefile) from templates.

The primary purpose is to replace autoconf-generated configure shell scripts in Gauche extension packages.

The advantage of using autoconf is that it generates a script that runs on most vanilla unix variants, for it only uses minimal shell features and basic unix commands. However, when you configure Gauche extension, you sure have Gauche already, so you don’t need to limit yourself with minimal environment.

Writing a configure script directly in Gauche means developers don’t need an extra step to generate configure before distribution. They can directly check-in configure in the source repo, and anybody who pulls the source tree can run configure at once without having autoconf.

Currently, gauche.configure only covers small subset of autoconf, though, so if you need to write complex tests you may have to switch back to autoconf. We’ll add tests as needed.

The core feature of gauche.configure is the ability to generate files (e.g. Makefile) from templates (e.g. Makefile.in) with replacing parameters. We follow autoconf convention, so the substitution parameters in a template is written like VAR@. You should be able to reuse Makefile.in used for autoconf without changing them.

The API corresponds to autoconf’s AC_* macros, while we use cf- prefix instead.

9.8.1 Structure of configure script and build files

A configure script tests running system’s properties to determine values of substitution parameters, then read one or more template build files, and write out one output build file for each, replacing substitution parameters for the assigned values.

By convention, a template file has a suffix .in, and the corresponding output file is named without the suffix. For example, Makefile.in is a template that generates Makefile.

Templates may contain substitution parameters, noted @PARAMETER_NAME@. This is a fragment of a typical Makefile template:

GAUCHE_PACKAGE = "@GAUCHE_PACKAGE@"
SOEXT = @SOEXT@
LOCAL_PATHS = "@LOCAL_PATHS"

foo.$(SOEXT): $(foo_SRCS)
   $(GAUCHE_PACKAGE) compile \ 
   --local=$(LOCAL_PATHS) --verbose foo $(foo_SRCS)

When processed by configure, @GAUCHE_PACKAGE@, @SOEXT@ and @LOCAL_PATHS@ are replaced with appropriate values. If you know autoconf, you are already familiar with this.

The Gauche configure script is structurally similar to autoconf’s configure.in, but you can use full power of Scheme. The following is the minimal configure script:

```scheme
#!/usr/bin/env gosh
(use gauche.configure)
(cf-init-gauche-extension)
(cf-output-default)
```

This script does several common tasks. The `cf-init-gauche-extension` does the following:

- First, it handles command-line arguments given to configure. In the default settings, it recognizes standard configure arguments such as `--prefix`, and `--with-local=PATH:PATH:...`. which adds PATH/includes and PATH/libs to the header and library search paths. You can handle more arguments by adding `cf-arg-with` and `cf-arg-enable` before `cf-init-gauche-extension`.
- Then it reads `package.scm`. Package name and version are taken from it. Dependencies are also checked.
- It sets up global environment to run other configure checks.
- It sets up default values for standard substitution parameters such as `@prefix@`.

And `cf-output-default` does the following:

- Generate gpd (Gauche package description) file.
- Writes package version to `VERSION` file.
- Scan `Makefile.in`'s in the source directory and its subdirectories, and process them to generate `Makefile`. If config header files (typically `config.h`) are specified by `cf-config-headers`, process input files (e.g. `config.h.in`) to generate the header files.

In general, a `configure` script consists of the following parts:

1. Extra argument declarations (optional): Declare `--with-PACKAGE` and/or `--enable-FEATURE` options you want to handle, by `cf-with-arg` and `cf-enable-arg`, respectively.
2. Initialization. Call to `cf-init` or `cf-init-gauche-extension` sets up global context and parses command-line arguments passed to configure. It also process package metainformation in `package.scm`, if it exists.
3. Tests and other substitution parameter settings (optional): Check system characteristics and sets up substitution parameters and/or C preprocessor definitions.
4. Output generation. Call `cf-output` or `cf-output-default` to process template files.

Most `cf-*` API corresponds to autoconf’s `AC_*` or `AS_*` macros. We need argument declarations before `cf-init` so that it can generate help message including custom arguments in one pass.

### 9.8.2 Configure API
## Initialization

**cf-init-gauche-extension**

This is a convenience API that packages several boilerplate `cf-*` function calls in one call. This must be called exactly once in a configure script.

Specifically, it calls `cf-arg-with` to process `--with-local`, then calls `cf-init` with no arguments to initialize, then sets the following substitution parameters:

- **GOSH** Path to `gosh`.
- **GAUCHE_CONFIG** Path to `gauche-config`.
- **GAUCHE_PACKAGE** Path to `gauche-package`.
- **GAUCHE_INSTALL** Path to `gauche-install`.
- **GAUCHE_CESCONV** Path to `gauche-cesconv`.
- **GAUCHE_PKGINCDIR** Result of `gauche-config --pkgincdir`.
- **GAUCHE_PKGLIBDIR** Result of `gauche-config --pkglibdir`.
- **GAUCHE_PKGARCHDIR** Result of `gauche-config --pkgarchdir`.

The `--with-local` command-line argument should take a parameter, which is a colon-separated list of paths. It becomes a value of substitution parameter `LOCAL_PATHS`. The default `Makefile.in` template passes its value to `gauche-package` via the `--local` argument. When the C extension is compiled and linked, the `include` and `lib` subdirectories of the given paths are searched for headers and libraries, respectively. For example, if the extension requires a library foo and you install it under `/opt/foo` (that is, headers in `/opt/foo/include` and library objects in `/opt/foo/lib`), then you can pass `--with-local=/opt/foo` to configure the extension.

If you don’t like these default behavior, you can call individual `cf-*` functions instead. See `cf-init` below.

**cf-init** :optional `package-name package-version maintainer-email`

This is a convenience API that initializes the configure system. Corresponds to autoconf’s `AC_INIT`. This must be called once in the configure script, before any feature-test procedures. (If you call `cf-init-gauche-extension`, `cf-init` is called from it.)

First, it checks if a file named `package.scm` is in the same directory as the configure script, and reads the Gauche package description from it. The package description contains package name, version, dependencies, etc. See Section 9.23 [Package metainformation], page 477, for the details.

It then parses the command-line arguments, sets up the configure environment, and (if `package.scm` defines dependencies) check if the system has required packages.

The optional arguments are only supported for the backward compatibility if you don’t have `package.scm`. You need at least to provide `package-name` and `package-version` to tell what package you’re configuring. They are used as the value of substitution parameter
PACKAGE_NAME and PACKAGE_VERSION. The other optional arguments, maintainer-email and homepage-url, are used to initialize PACKAGE_BUGREPORT and PACKAGE_URL. These arguments are compatible to autoconf’s AC_INIT macro.

We recommend to always use package.scm and omit all the optional arguments, because it allows you to maintain the package metainformation in one place. When package.scm is read, PACKAGE_BUGREPORT is initialized by the first entry of maintainers slot of the package description, and PACKAGE_URL is initialized by its homepage slot. See Section 9.23 [Package metainformation], page 477, for description of slots of the package description.

Note that if there’s package.scm and you provide the optional arguments, they must match, or cf-init raises an error. It is to catch errors during transition in which you forgot to update either one.

This procedure sets up a bunch of standard substitution parameters such as prefix, bindir or srcdir. To see what substitution parameters are set, you can call cf-show-substs after cf-init.

Command-line arguments

cf-arg-enable feature help-string :optional proc-if-given
  proc-if-not-given

 cf-arg-with package help-string :optional proc-if-given proc-if-not-given
 {gauche.configure} Make the configure script accept feature selection argument and package selection argument, respectively. The corresponding autoconf macros are AC_ARG_ENABLE and AC_ARG_WITH.

Those procedures must be called before calling cf-init or cf-init-gauche-extension.

The feature and package arguments must be a symbol.

A feature selection argument is in a form of either --enable-feature=val, --enable-feature, or --disable-feature. The latter two are equivalent to --enable-feature=yes and --enable-feature=no, respectively. It is to select an optional feature provided with the package itself.

A package selection argument is in a form of either --with-package=val, --with-package and --without-package. The latter two are equivalent to --with-package=yes and --with-package=no, respectively. It is to select an external software package to be used with this package.

When cf-init finds these arguments, it adds entry of feature or package to the global tables, with the value val. Those global tables can be accessed with cf-feature-ref and cf-package-ref procedures below.

The help-string argument must be a string and is used as is to list the help of the option in part of usage message displayed by configure --help. You can use cf-help-string below to create a help string that fits nicely in the usage message.

If optional proc-if-given argument is given, it must be a procedure that accepts one argument, val. It is called when cf-init finds one of those arguments.

If optional proc-if-not-given argument is given, it must be a procedure that accepts no arguments. It is called when cf-init doesn’t find any of those arguments.

The following is to accept --with-local=PATH:PATH:... (This cf-arg-with call is included in cf-init-gauche-extension). Note that the help string (the second argument) is generated by cf-help-string call. The command-line parameter followed by --with-local is passed as the argument of the procedure in the third argument:

  (cf-arg-with 'local
   (cf-help-string
"--with-local=PATH:PATH..."
"For each PATH, add PATH/include to the include search paths and PATH/lib to the library search paths. Useful if you have some libraries installed in non-standard places. ")

([with-local]
 (unless (member with-local '("yes" "no" "))
 (cf-subst 'LOCAL_PATHS with-local))
 (~[] (cf-subst 'LOCAL_PATHS "")))

**cf-help-string**  
*item description*  

{gauche.configure} Return a string formatted suitable to show as an option’s help message. The result can be passed to help-string argument of cf-arg-enable and cf-arg-with. This corresponds to autoconf’s AS_HELP_STRING.  

Call it as follows, and it’ll indent and fill the description nicely.

(cf-help-string "--option=ARG" "Give ARG as the value of option")

**cf-feature-ref**  
*name*  

{gauche.configure} Lookup a symbol *name* from the global feature table and the global package table, respectively. These can be called after cf-init.  

For example, if you’ve called cf-arg-enable with foofeature, and the user has invoked the configure script with --with-foofeature=full, then (cf-feature-ref 'foofeature) returns "full". If the user hasn’t given the command-line argument, #f is returned.  

If you add or change the value of features or packages, you can use generalized set! as (set! (cf-feature-ref 'NAME) VALUE) etc.

**Messages**

The cf-init procedure opens the default log drain that goes to config.log, and you can use log-format to write to it (See Section 9.17 [User-level logging], page 458, for the details of logging).  

However, to have consistent message format conveniently, the following procedures are provided. They emits the message both to log files and the current output port (in slightly different formats so that the console messages align nicely visually.)

**cf-msg-checking**  
*funt arg ...*  

{gauche.configure} Writes out “checking XXX...” message. The *funt* and *arg ...* arguments are passed to format to produce the “XXX” part (see Section 6.21.8.4 [Formatting output], page 279).  

For the current output port, this does not emit the trailing newline, expecting cf-msg-result will be called subsequently.  

Here’s an excerpt of the source that uses cf-msg-checking and cf-msg-result:

```
(define (compiler-can-produce-executable?)
 (cf-msg-checking "whether the ~a compiler works" (~ (cf-lang)'name))
 (rlet1 result ($ run-compiler-with-content
 (cf-lang-link-m (cf-lang))
 (cf-lang-null-program-m (cf-lang)))
 (cf-msg-result (if result "yes" "no"))))
```

This produces a console output like this:

```
checking whether the C compiler works... yes
```

while the log file records more info:

```
checking: whether the C compiler works
```
... whatever logging message from run-compiler-with-content ...
result: yes

This corresponds to autoconf’s AC_MSG_CHECKING.

cf-msg-result fmt arg . . . [Function]
{gauche.configure} The fmt and arg . . . are passed to format, and the formatted message
and newline are written out (see Section 6.21.8.4 [Formatting output], page 279). For the log
file, it records “result: XXX” where XXX is the formatted message. Supposed to be used
with cf-msg-checking.

This corresponds to autoconf’s AC_MSG_RESULT.

cf-msg-notice fmt arg . . . [Function]
{gauche.configure} Produces formatted message to both console and log. Newline is added.
This corresponds to autoconf’s AC_MSG_NOTICE.

cf-msg-warn fmt arg . . . [Function]
cf-msg-error fmt arg . . . [Function]
{gauche.configure} Produces “Warning: XXX” and “Error: XXX” messages, respectively.
The fmt and arg . . . are passed to format to generate XXX part (see Section 6.21.8.4
[Formatting output], page 279). Then, cf-msg-error exits with exit code 1.

These corresponds to autoconf’s AC_MSG_WARN and AC_MSG_ERROR. NB: AC_MSG_ERROR can
specify the exit code, but cf-msg-error uses fixed exit code (1) for now.

cf-echo arg . . . [> file][>> file] [Function]
{gauche.configure} Convenience routine to replace shell’s echo command.

If the argument list ends with > file or >> file, where file is a string file name, then
this works just like shell’s echo: that is, args except the last two are written to file, space
separated, newline terminated. Using > supersedes file, while >> appends to it.

If the argument list doesn’t end with those redirection message, it writes out the argument
to both the current output port and the log file, space separated, newline terminated. For
the log file, the message is prefixed with “Message:”.

Parameters and definitions

The configure script maintains two global tables, definition tables and parameter tables. Definition
tables is used for C preprocessor definitions, and parameter tables are used for @PARAMETER@
substitutions. (Do not confuse substitution parameters with Scheme’s parameter objects (see
Section 6.16.1 [Parameters], page 233)).

cf-define symbol :optional value [Function]
{gauche.configure} Registers C preprocessor definition of symbol with value. Value
can be any Scheme objects, but it is emitted to a command line (in -DSYMBOL=VALUE form) or in
config.h (in #define SYMBOL VALUE form) using display, so you want to avoid including
funny characters. If value is omitted, 1 is assumed.

NB: To #define a string value, e.g. #define FOO "foo", you have to call as (cf-define
'FOO "\"foo\""').

This corresponds to autoconf’s AC_DEFINE.

cf-defined? symbol [Function]
{gauche.configure} Returns #t iff symbol is cf-defined.
**Chapter 9: Library modules - Gauche extensions**

**cf-subst symbol value**

{gauche.configure} Registers a substitution parameter `symbol` with `value`. Value can be any Scheme objects; its display representation is used to substitute `@SYMBOL@` in the template.

This corresponds to autoconf’s `AC_SUBST`, but we require the value (while autoconf can refer to the shell variable value as default).

**cf-subst-prepend symbol value :optional delim default**
**cf-subst-append symbol value :optional delim default**

{gauche.configure} Prepend or append `value` to the substitution parameter `symbol`, using delimiter `delim`. If the substitution parameter isn’t defined, `value` becomes the sole value of the parameter, except when `default` is given and not an empty string. If `delim` is omitted, single whitespace is used.

**with-cf-subst ((symbol value) ...) ...**

{gauche.configure} Temporarily replace substitution parameters with new values. This could be useful for example to run some compilation tests with different parameters

```scheme
(with-cf-subst ((LIBS "-L<path> -l<lib>")
    (cf-try-compile-and-link ...))
```

**cf-have-subst? symbol**

{gauche.configure} Returns true iff `symbol` is registered as a substitution parameter by `cf-subst`.

**cf-arg-var symbol**

{gauche.configure} Lookup the environment variable `symbol` and if it is found, use its value as the substitution value. For example, if you call `(cf-arg-var 'MYCFLAGS)` then the user can provide the value of `@MYCFLAGS@` as `MYCFLAGS=-g ./configure`.

This corresponds to autoconf’s `AC_ARG_VAR`, but we lack the ability of setting the help string. That’s because `cf-arg-var` must be run after `cf-init`, but the help message is constructed within `cf-init`.

**cf-ref symbol :optional default**

{gauche.configure} This looks up the value of the substitution parameter `symbol`. If there’s no such substitution parameter registered, it returns `default` when it’s provided, otherwise throws an error.

**cf$ symbol**

{gauche.configure} Looks up the value of the substitution parameter `cf-ref`, but it returns empty string if it’s unregistered. Useful to use within string interpolation, e.g. `#"gosh ~(cf$'GOSHFLAGS)"`.

**Predefined tests**

**cf-check-prog sym prog-or-progs :key value default paths filter**
**cf-path-prog sym prog-or-progs :key value default paths filter**

{gauche.configure} Check if a named executable program exists in search paths, and if it exists, sets the substitution parameter `sym` to the name of the found program. The name to search is specified by `prog-or-progs`, which is either a string or a list of strings.

The difference of `cf-check-prog` and `cf-path-prog` is that `cf-check-prog` uses the base-name of the found program, while `cf-path-prog` uses its full path. These corresponds to autoconf’s `AC_CHECK_PROG`, `AC_CHECK_PROGS`, `AC_PATH_PROG` and `AC_PATH_PROGS`. 
For example, the following feature test searches either one of cc, gcc, tcc or pcc in PATH and sets the substitution parameter MY_CC to the name of the found one.

(cf-check-prog 'MY_CC '("cc" "gcc" "tcc" "pcc")

If multiple program names is given, the search is done in the following order: First, we search for the first item (cc, in the above example) for each of paths, then the second, etc. For example, if we have /usr/local/bin:/usr/bin:/bin in PATH and we have /usr/local/bin/tcc and /usr/bin/gcc, the above feature test sets MY_CC to "gcc". If you use cf-path-prog instead, MY_CC gets "/usr/bin/gcc".

If no program is found, sym is set to the keyword argument default if it is given, otherwise sym is left unset.

If the value keyword argument is given, its value is used instead of the found program name to be set to sym.

The list of search paths is taken from PATH environment variable. You can override the list by the paths keyword argument, which must be a list of directory names. It may contain nonexistent directory names, which are silently skipped.

The filter keyword argument, if given, must be a predicate that takes full pathname of the executable program. It is called when the procedure finds matching executable; the filter procedure may reject it by returning #f, in which case the procedure keeps searching.

Note: If the substitution parameter sym is already set at the time these procedure is called, these procedures do nothing. Combined with cf-arg-var, it allows the configure script caller to override the feature test. For example, suppose you have the following in the configure script:

(cf-arg-var 'GREP)
(cf-path-prog 'GREP '("egrep" "fgrep" "grep")

A user can override the test by calling configure like this:

$ ./configure GREP=mygrep

cf-prog-cxx
{gauche.configure} A convenience feature test to find C++ compiler. This searches popular names of C++ compilers from the search paths, sets the substitution parameter CXX to the compiler’s name, then tries to compile a small program with it to see it can generate an executable.

This corresponds to autoconf’s AC_PROG_CXX.

CXX is cf-arg-var’ed in this procedure. If a user provide the value when he calls configure, the searching is skipped, but the check of generating an executable is still performed.

If the substitution parameter CXXFLAGS is set, its value is used to check if the compiler can generate an executable. CXXFLAGS is cf-arg-var’ed in this procedure.

This procedure also emulates autoconf’s AC_PROG_CXX behavior— if CXX is not set, but CCC is set, then we set CXX by the value of CCC and skip searching.

cf-header-available? header :key includes
{gauche.configure} Check if a header file header exists and usable, by compiling a source program of the current language that includes the named header file. Return #t if the header is usable, #f if not.

Both procedure does the same thing. The name cf-check-header corresponds to autoconf’s AC_CHECK_HEADER.

If header requires other headers being included or preprocessor symbols defined before it, you can pass a list of strings to be emitted before the check in the includes keyword arguments.
The given strings are just concatenated and used as a C program fragment. The default value is provided by \texttt{cf-includes-default}.

The following example sets C preprocessor symbol \texttt{HAVE\_CRYPT\_H} to 1 if \texttt{crypt.h} is available. (Note: For this kind of common task, you can use \texttt{cf-check-headers} below. The advantage of using \texttt{cf-check-header} is that you can write other actions in Scheme depending on the result.)

\begin{verbatim}
(when (cf-check-header "crypt.h")
  (cf-define "HAVE\_CRYPT\_H" 1))
\end{verbatim}

\texttt{cf-check-headers} \texttt{headers :key includes if-found if-not-found} 

\begin{verbatim}
{gauche.configure} Codify a common pattern of checking the availability of headers and sets C preprocessor definitions. This corresponds to autconf's \texttt{AC\_CHECK\_HEADERS}. This procedure is invoked for the side effects, and returns an undefined value.

See this example:

\begin{verbatim}
(cf-check-headers '("unistd.h" "stdint.h" "inttypes.h" "rpc/types.h"))
\end{verbatim}

This checks availability of each of listed headers, and sets C preprocessor definition \texttt{HAVE\_UNISTD\_H\_H}, \texttt{HAVE\_STDINT\_H\_H}, \texttt{HAVE\_INTTYPES\_H\_H} and \texttt{HAVE\_RPC\_TYPES\_H\_H} to 1 if the corresponding header file is available.

A list of strings given to \texttt{includes} are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, \texttt{cf-includes-default} provides the default list of such headers.

The keyword argument \texttt{if-found} and \texttt{if-not-found} are procedures to be called when a header is found to be available or to be unavailable, respectively. The procedure receives the name of the header.

The name of the C preprocessor definition is derived from the header name by upcasing it and replacing non-alphanumeric characters for \texttt{\_}. Note that this substitution is not injective: Both \texttt{gdbm/ndbm.h} and \texttt{gdbm-ndbm.h} yield \texttt{GDBM\_NDBM\_H}. If you need to distinguish such files you have to use \texttt{cf-check-header}.

\texttt{cf-includes-default} \texttt{includes if-found if-not-found} 

\begin{verbatim}
{gauche.configure} Returns a list of strings that are included in the check program by default. It is actually a combination of C preprocessor \#ifdefs and \#includes, and would probably be better to be called \texttt{cf-prologue-default} or something, but the corresponding autoconf macro is \texttt{AC\_INCLUDES\_DEFAULT} so we stick to this name.

Usually you don't need to call this explicitly. Not giving the \texttt{includes} argument to \texttt{cf-check-header} and \texttt{cf-check-headers} will make \texttt{cf-includes-default} called implicitly.

\texttt{cf-type-available?} \texttt{type :key includes} 

\begin{verbatim}
{gauche.configure} Test if \texttt{type} is defined as a type name. Return \#t if \texttt{type} is defined, \#f otherwise.

Two procedures are the same. The name \texttt{cf-check-type} corresponds to \texttt{AC\_CHECK\_TYPE}.

A list of strings given to \texttt{includes} are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, \texttt{cf-includes-default} provides the default list of such headers.

\texttt{cf-check-types} \texttt{types :key includes if-found if-not-found} 

\begin{verbatim}
{gauche.configure} For each type in the list \texttt{types}, call \texttt{cf-check-type} to see it is defined as a type. If it is, defines \texttt{HAVE\_\texttt{type}}, and calls \texttt{if-found} with the type as an argument if provide. If the type is not defined and \texttt{if-not-found} is provided, calls it with the type as an argument.
\end{verbatim}
The argument *includes* is passed to `cf-check-type`.

This corresponds to autoconf’s `AC_CHECK_TYPES`.

Returns an undefined value. This procedure is for side effects.

```plaintext
;; May define HAVE_PTRDIFF_T and/or HAVE_UNSIGNED_LONG_LONG_INT
;; depending on its availability:
(cf-check-types '("ptrdiff_t" "unsigned long long int"))

(cf-check-types '("float_t")
  :includes '((,@(cf-includes-default)
    "#include <math.h>
    "include <math.h>\n"))
```

**Function**

`cf-decl-available? symbol :key includes` [Function]

`cf-check-decl symbol :key includes` [Function]

{gauche.configure} Test if `symbol` is declared as a cpp macro, a variable, a constant, or a function. Return \#t if type is defined, \#f otherwise.

Two procedures are the same. The name `cf-check-decl` corresponds to autoconf’s `AC_CHECK_DECL`.

A list of strings given to *includes* are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, `cf-includes-default` provides the default list of such headers.

**Function**

`cf-check-decls symbols :key includes if-found if-not-found` [Function]

{gauche.configure} For each symbol in `symbols`, call `cf-check-decl` to see if it is declared. If it is, define `HAVE_DECL_symbol` to 1, and calls `if-found` with the symbol if provided. If it is not declared, define `HAVE_DECL_symbol` to 0, and calls `if-not-found` with the symbol if provided. This corresponds to autoconf’s `AC_CHECK_DECLS`.

The argument *includes* is passed to `cf-check-decl`.

This procedure returns an undefined value. This procedure is for side effects.

Note that, unlike other `cf-*check-*` routines which leave `HAVE-*` macro undefined when the item isn’t found, this one always defines the macro and differentiate the result with its value.

This behavior is the same as `AC_CHECK_DECLS`.

**Function**

`cf-member-available? aggregate.member :key includes` [Function]

`cf-check-member aggregate.member :key includes` [Function]

{gauche.configure} The `aggregate.member` argument is a string of aggregate type name and its member concatenated by a dot, e.g. "struct password.pw_gecos". It can also be a submember, e.g. "struct foo.bar.baz". The `aggregate` part can be any type name (typedef-ed name is ok).

This test checks if `member` is a member of `aggregate`, and returns \#t if so, or returns \#f if not.

Two procedures are the same. The name `cf-check-member` corresponds to autoconf’s `AC_CHECK_MEMBER`.

A list of strings given to *includes* are emitted to the C source file before the inclusion of the testing header. You can give necessary headers and/or C preprocessor definitions there; if omitted, `cf-includes-default` provides the default list of such headers.

**Function**

`cf-check-members members :key includes if-found if-not-found` [Function]

{gauche.configure} For each `aggregate.member` in `members`, call `cf-check-member`. If the test passes, defines `HAVE_aggregate.member` in `members`, and calls `if-found` with `aggregate.member` if provided. If the test fails, calls `if-not-found` with `aggregate.member` if provided.
This corresponds to autoconf’s `AC_CHECK_MEMBERS`.

The `include` argument is passed to `cf-check-member`.

```scheme
;; Defines HAVE_STRUCT_ST_RDEV and/or HAVE_STRUCT_ST_BLKSIZE
;; depending on their availability:
(cf-check-members '("struct stat.st_rdev"
    "struct stat.st_blksize")

This procedure is for side effects, and returns an undefined value.

`cf-func-available? func`  [Function]

`cf-check-func func`  [Function]

``gauche.configure``) See if a function `func` is available. This emits C code to call `func` (with dummy declaration) and tries to compile and link, using current value of substitution parameter `LIBS`. The value of `cf-includes-default` is at the top of the emitted C code.

They return `#t` if `func` is available, `#f` otherwise.

Two procedures are the same. The name `cf-check-func` corresponds to autoconf’s `AC_CHECK_FUNC`.

`cf-check-funcs funcs :key if-found if-not-found`  [Function]

``gauche.configure``) For each function name `func` in `fun cs`, call `cf-check-func` to determine availability. If it is available, define HAVE_func, and calls `if-found` with `func` if provided. If it is not available, calls `if-not-found` with `func` if provided.

This corresponds to autoconf’s `AC_CHECK_FUNCS`.

This procedure is for side effects, and returns an undefined value.

`cf-lib-available? lib fn :key other-libs if-found if-not-found`  [Function]

`cf-check-lib lib fn :key other-libs if-found if-not-found`  [Function]

``gauche.configure``) See if a library `lib` can be linked and a function `fn` in it is callable. Return `#t` if it is, `#f` if not.

Two procedures are the same. The name `cf-check-lib` corresponds to autoconf’s `AC_CHECK_LIB`.

Give the name you pass after `-l` option to `lib`; for example, if you want to check availability of `libm`, you can say as follows:

```
(cf-check-lib "m" "sin")
```

This generates a C source that calls `fn` and try to compile and link it to generate executable. If linking `lib` requires additional libraries, it should be listed in `other-libs`:

```
(cf-check-lib "Xt" "XtDisplay" :other-libs '("-lX11" "-lSM" "-lICE")
```

If compilation and linking succeeds, `if-found` is called at the tail position with the library name (`"m"` and `"Xt"` in the above examples, respectively) as the argument. The default behavior is to add `-l`lib in the left of substitution parameter `LIBS`, and set `HAVE_LIBlib` definition, then returns `#t`.

If compilation or linking fails, `if-not-found` is called at the tail position with the library name. The default behavior is to return `#f`.

The default behavior of `if-found` and `if-not-found` allows `cf-check-lib` to be used as predicate as well. If you merely want to take an action depending on whether the library is found or not, you can write like this:

```
(unless (cf-check-lib "foo" "foo_fn"
    ... do something if libfoo isn’t available ...)
```

Use `if-found` and/or `if-not-found` only if you want to override the default behaviors.
cf-search-libs  fn libs :key other-libs if-found if-not-found  
{gauche.configure} Like cf-check-lib, but can be used if you’re not sure which library contains desired function. This corresponds to autoconf’s AC_SEARCH_LIBS. Note that this takes function name first, while cf-check-lib takes function name second—blame autoconf for this inconsistency.

First it tests if fn is available without any library in libs (that is, with the ones already in LIBS and specified in other-libs). If not, it tests each library in libs in turn.

If fn is found, if-found is called at the tail position, with the name of the library as an argument (if fn is available without any library, the argument is #f). If omitted, and a library is required, then the library is added to the substitution parameter LIBS. The default procedure returns #t.

If fn isn’t found in any of the libraries, if-not-found is called at the tail position with #f as the argument. The default procedure does nothing and just returns #f.

The default behavior of if-found and if-not-found allows cf-search-libs to be used as predicate as well. If you give alternative procedures, keep in mind that their return value will be returned to cf-search-libs.

Running compiler

The gauche.configure module provides a generic mechanism to construct a small test program, compile it, and run it. Currently we only support C and C++; we’ll add support for other languages as needed.

cf-lang  
{gauche.configure}

cf-lang-program  prologue body  
{gauche.configure} Returns a string tree that consists a stand-alone program for the current language. Prologue and body must be a string tree. Prologue comes at the beginning of the source, and body is included in the part of the program that’s executed. If the current language is C, the code fragment:

(use text.tree)
(write-tree (cf-lang-program "#include <stdio.h>
" "printf(\"()\")\n")
)

would produce something like this:

#include <stdio.h>

int main(){
  printf("()");
  ; return 0;
}

cf-lang-io-program  
{gauche.configure} This is a convenience routine. It returns a string tree of a program in the current language, that creates a file named conftest.out, then exits with zero status on success, or nonzero status on failure.

cf-lang-call  prologue func-name  
{gauche.configure}

cf-try-compile  prologue body  
{gauche.configure}
Chapter 9: Library modules - Gauche extensions

**cf-try-compile-and-link prologue body**

{gauche.configure}

Output

**cf-output-default file ...**

{gauche.configure} A convenience routine to produce typical output. It does the following:

- Generate gpd (Gauche package description) file using cf-make-gpd.
- Generate VERSION file that contains the value of PACKAGE_VERSION substitution parameter.
- Search Makefile.in's under the source directory (the value of substitution parameter srcdir), and process them to produce Makefile's. If file ... are given, file.in are also processed as well to produce file.

See cf-output below for the details.

- If config header is registered by cf-config-headers, process them as well.

**cf-output file ...**

{gauche.configure} Generates file's from the input templates. This corresponds to autoconf's AC_OUTPUT.

For each file, a file named file.in is read as a template. Within the file, @PARAMETER@ is substituted with the value of (cf$ 'PARAMETER). If the named parameter isn't registered, a warning is issued and the parameter is left unsubstituted.

If config headers are not registered via cf-config-headers, a substitution parameter DEFS is replaced with all the definitions in the form of -D... For example, if you have checked header files foo/bar.h and foo/baz.h, DEFS gets the value -DHAVE_FOO_BAR_H -DHAVE_FOO_BAZ_H.

If config header is registered by cf-config-headers, they are processed as well. In such case, the substitution parameter DEFS gets the value -DHAVE_CONFIG_H.

**cf-config-headers header-or-headers**

{gauche.configure} Sets up config header files to be processed. Usually a config header file is named config.h, and contains definitions determined by feature tests.

The header-or-headers argument may be a string header-spec or a list of string header-specs, where each header spec is a header file name (e.g. "config.h") or a header name and a input file name concatenated with a colon (e.g. "config.h:config.h.templ"). If it's just a header name, input file name is assumed to be the header file name with ".in" appended.

The input template of config header file contains a bunch of #undef directives, such as the following:

```c
/* Gauche ABI version string */
#undef GAUCHE_ABI_VERSION

/* Define if Gauche handles multi-byte character as EUC-JP */
#undef GAUCHE_CHAR_ENCODING_EUC_JP

/* Define if Gauche handles multi-byte character as Shift JIS */
#undef GAUCHE_CHAR_ENCODING_SJIS

/* Define if Gauche handles multi-byte character as UTF-8 */
#undef GAUCHE_CHAR_ENCODING_UTF_8
```

Once processed, the generated header file has either #undef line is replaced with #define, or commented out, depending on the definitions determined by feature tests.

```c
/* Gauche ABI version string */
```
#define GAUCHE_ABI_VERSION "0.97"

/* Define if Gauche handles multi-byte character as EUC-JP */
/* #undef GAUCHE_CHAR_ENCODING_EUC_JP */

/* Define if Gauche handles multi-byte character as Shift JIS */
/* #undef GAUCHE_CHAR_ENCODING_SJIS */

/* Define if Gauche handles multi-byte character as UTF-8 */
#define GAUCHE_CHAR_ENCODING_UTF_8 /**/

Note that the lines other than #undef are copied as they are.
The substitution parameter DEFS behaves differently whether config header is specified or not. If no config header is registered, The value of DEFS is a C command-line arguments for definitions, e.g. `-DGAUCE_ABI_VERSION=0.97` `-DGAUCE_CHAR_ENCODING_UTF8`. If config header files are registered, the value of DEFS becomes simply `-DHAVE_CONFIG_H`.

cf-show-substs :key formatter
    {gauche.configure} Print all substitution parameters; this is for debugging.
    For each substitution parameter name and value, formatter is called with them; the default is (`[^k v] (format #t "\"16s \"s\" k v))`.

cf-make-gpd
    {gauche.configure} Generate gpd (Gauche package description) file, PACKAGE_NAME.gpd, where PACKAGE_NAME is the package's name either taken form package.scm or the argument to cf-init. See Section 9.23 [Package metainformation], page 477, for the package description file format.

9.9 gauche.connection - Connection framework

gauce.connection
    A connection is an abstract class to handle full-duplex communication channel. The actual data I/O is done through Scheme ports; a channel provides an interface to retrieve those ports. It also has interface to know endpoint names, and the way to shutdown the communication.
The <socket> class, for example, implements the connection framework (see Section 9.22 [Networking], page 464). The <tls> class, which wraps a socket to provide TLS communication, also implements it. That means you can write a client code that works both plain socket connection and secure connection. You can also abstract communication to the external process as a connection (see Section 9.27.5 [Process connection], page 501).

Each of the connection endpoints (self for our side, and peer to the other side) has addresses, some object that identifies the endpoint. The framework doesn't specify the actual type of addresses; it only requires that addresses can be passed to connection-address-name method to get its string representation, so that can be used for logging and monitoring. The concrete class can choose suitable address representation. For example, <socket> uses <sockaddr> for the addresses.

A concrete class must implement the following methods.

    connection-self-address (c <connection>)
    connection-peer-address (c <connection>)
    connection-input-port (c <connection>)
    connection-output-port (c <connection>)
    connection-shutdown (c <connection>) how
    connection-close (c <connection>)
At this moment this framework doesn’t provide a generic way to create a connection, since the way to do it may greatly vary depending on the concrete implementation. Each concrete implementation should provide its own procedure to create and return a new connection.

``` Scheme
connection-self-address connection ; [Method]
connection-peer-address connection ; [Method]

Returns the address of this connection’s endpoint and its peer’s. For sockets and <tls>, it is an instance of `<sockaddr>`. For processes, it is a string describing the process.

If connection is not connected, these methods can return #f.

The returned value (other than #f) must be able to be passed to connection-address-name method.

Currently addresses are only used for logging purpose within the connection framework; however, we may enhance the framework to add “connect” operation, for example, so the concrete class is encouraged to use objects that can be used to create connections.

``` Scheme
connection-address-name address ; [Method]

This method returns a string representation of address, which is a returned value from connection-self-address and connection-peer-address. The default method just uses address’s display-representation. If a concrete class chooses aggregate objects to represent addresses, it should provide this method as well.

``` Scheme
connection-input-port connection ; [Method]
connection-output-port connection ; [Method]

Returns an input port and an output port to read from and write to the connection, respectively.

If the connection is not connected, or already shutdown or closed, the return value of these methods are unspecified.

``` Scheme
connection-shutdown connection :optional how ; [Method]

Shutdown the connection. You can either shutdown the connection entirely at once, or shutdown only read or write channel of it.

Shutdown is about telling the peer to terminate the communication. Usually the peer will detect the termination by reading EOF from their input channel. The port corresponding to the shutdown channel is closed so no further communication is possible.

Note that merely closing the connection doesn’t shutdown the connection—the process may fork after creating the connection, and in that case, one process may close the connection without shutting down.

The how argument must be one of those symbols:

- read Shutdown read channel of the connection.
- write Shutdown write channel of the connection.
- both Shutdown both channels of the connection.

If omitted, both is assumed.

Shutdown down already shut down channel has no effect.

Note: Some concrete implementation of connection may not allow to shutdown each channels independently. In such case, the connection is shut down entirely regardless of how argument. The one-argument case is handled by the default method (it calls two-argument method with both as how). So the concrete class only need to define two argument method.
connection-close connection  [Method]
Close the connection. This destroys the connection and frees local resources. Note that this does not shutdown the connection itself. If this connection is the only endpoint of this side, the peer will get an error when it tries to communicate.

See connection-shutdown for the details of shutting down a connection.

9.10 gauche.dictionary - Dictionary framework

gauche.dictionary  [Module]
A dictionary is an abstract class for objects that can map a key to a value. This module provides some useful generic functions for dictionaries, plus generic dictionary classes built on top of other dictionary classes.

9.10.1 Generic functions for dictionaries

These generic functions are useful to implement algorithms common to any dictionary-like objects, a data structure that maps discrete, finite set of keys to values. (Theoretically we can think of continuous and/or infinite set of keys, but implementation-wise it is cleaner to limit the dictionary

Among built-in classes, <hash-table> and <tree-map> implement the dictionary interface. All the <dbm> classes provided by dbm module also implement it.

To make your own class implement the dictionary interface, you have to provide at least dict-get, dict-put!, dict-delete!, dict-fold and dict-comparator. (You can omit dict-delete! if the datatype doesn’t allow deleting entries.) Other generic functions have default behavior built on top of these. You can implement other methods as well, potentially to gain better performance.

(Note: Dictionaries are also collections, so you can use collection methods as well; for example, to get the number of entries, just use size-of).

dict-get (dict <dictionary>) key :optional default  [Generic function]
{gauche.dictionary} Returns the value corresponding to the key. If the dictionary doesn’t have an entry with key, returns default when it is provided, or raises an error if not.

dict-put! (dict <dictionary>) key value  [Generic function]
{gauche.dictionary} Puts the mapping from key to value into the dictionary.

(setter dict-get) (dict <dictionary>) key value  [Generic function]
{gauche.dictionary} This works the same as dict-put!.

dict-exists? (dict <dictionary>) key  [Generic function]
{gauche.dictionary} Returns #t if the dictionary has an entry with key, #f if not.

dict-delete! (dict <dictionary>) key  [Generic function]
{gauche.dictionary} Removes an entry with key form the dictionary. If the dictionary doesn’t have such an entry, this function is noop.

dict-clear! (dict <dictionary>)  [Generic function]
{gauche.dictionary} Empties the dictionary. Usually this is much faster than looping over keys to delete them one by one.

dict-comparator (dict <dictionary>)  [Generic function]
{gauche.dictionary} Should return a comparator used to compare keys.
dict-fold (dict <dictionary>) proc seed

{gauche.dictionary} Calls a procedure proc over each entry in a dictionary dict, passing a
seed value. Three arguments are given to proc: an entry’s key, an entry’s value, and a seed
value. Initial seed value is seed. The value returned from proc is used for the seed value of
the next call of proc. The result of the last call of proc is returned from dict-fold.

If dict is <ordered-dictionary>, proc is called in the way to keep the following associative
order, where the key is ordered from \( K_0 \) (minimum) to \( K_n \) (maximum), and the corresponding
values is from \( V_0 \) to \( V_n \):

\[
(\text{proc } K_n V_n (\text{proc } K_{n-1} V_{n-1} \ldots (\text{proc } K_0 V_0 \text{ seed})))
\]

dict-fold-right (dict <ordered-dictionary>) proc seed

{gauche.dictionary} Like dict-fold, but the associative order of applying proc is reversed
as follows:

\[
(\text{proc } K_0 V_0 (\text{proc } K_1 V_1 \ldots (\text{proc } K_n V_n \text{ seed})))
\]

This generic function is only defined on <ordered-dictionary>.

dict-for-each (dict <dictionary>) proc

{gauche.dictionary} Calls proc with a key and a value of every entry in the dictionary
dict. For ordered dictionaries, proc is guaranteed to be called in the increasing order of keys.

dict-map (dict <dictionary>) proc

{gauche.dictionary} Calls proc with a key and a value of every entry in the dictionary
dict, and gathers the result into a list and returns it. For ordered dictionaries, the result is
in the increasing order of keys (it doesn’t necessarily mean proc is called in that order).

dict-keys (dict <dictionary>)

{gauche.dictionary} Returns a list of all keys or values of a dictionary dict, respectively.
For ordered dictionaries, the returned list is in the increasing order of keys.

dict-values (dict <dictionary>)

{gauche.dictionary} Returns a list of pairs of key and value in the dictionary. The order
of pairs is undefined.

dict-push! (dict <dictionary>) key value

{gauche.dictionary} A shorthand way to say (dict-put! dict key (cons value (dict-
get dict key '()))). A concrete implementation may be more efficient (e.g. it may not
search key twice.)

dict-pop! (dict <dictionary>) key :optional fallback

{gauche.dictionary} If (dict-get dict key) is a pair \( p \), the entry value is replaced with
(cdr \( p \)) and the procedure returns (car \( p \)). If no entry for key is in the table, or the entry
isn’t a pair, the table isn’t modified, and fallback is returned if given, or an error is raised.

dict-update! (dict <dictionary>) key proc :optional fallback

{gauche.dictionary} Works like the following code, except that the concrete implementation
may be more efficient by looking up key only once.

\[
(\text{rlet1 } x (\text{proc } (\text{dict-get dict key fallback})))
\]

(define-method dict-put! ((dict <my-dict>) key value)

\[
(\text{dict-put! } \text{dict } \text{key } \text{x})
\]

define-dict-interface dict-class method proc method2 proc2 ... [Macro]

{gauche.dictionary} Many dictionary-like datatypes already has their own procedures that
directly corresponds to the generic dictionary API, and adding dictionary interface tends to
become a simple repetition of define-methods, like this:

\[
(\text{define-method dict-put! } ((\text{dict } <\text{my-dict}>) \text{key } \text{value})
(my-dict-put! key value))

The **define-dict-interface** macro is a convenient way to define those methods in a batch. Each *method* argument is a keyword that corresponds to **dict-method**, and *proc* is the name of the datatype-specific procedure. Here’s the definition of dict interface for `<tree-map>` and you’ll get the idea. You don’t need to provide every dictionary interface.

```
(define-dict-interface <tree-map>
 :get tree-map-get
 :put! tree-map-put!
 :delete! tree-map-delete!
 :clear! tree-map-clear!
 :comparator tree-map-comparator
 :exists? tree-map-exists?
 :fold tree-map-fold
 :fold-right tree-map-fold-right
 :for-each tree-map-for-each
 :map tree-map-map
 :keys tree-map-keys
 :values tree-map-values
 :pop! tree-map-pop!
 :push! tree-map-push!
 :update! tree-map-update!
 :->alist tree-map->alist)
```

### 9.10.2 Generic dictionaries

**Bimap**

*Class* `<bimap>`

`gauche.dictionary` Provides a bidirectional map (*bimap*), a relation between two set of values, of which you can lookup both ways.

Internally, a bimap consists of two dictionaries, *left* map and *right* map. Think a bimap as a relation between *x* and *y* s. The left map takes an *x* as a key and returns corresponding *y* as its value. The right map takes an *y* as a key and returns corresponding *x* as its value.

Currently, `<bimap>` only supports strict one-to-one mapping. Mutating interface (*bimap-*put!, *bimap-*delete! etc) modifies both left and right maps to maintain this one-to-one mapping. (In future, we may provide an option to make many-to-one and many-to-many mappings).

A bimap can be used as a dictionary, with the generic dictionary functions such as **dict-get**. In such cases, the left map takes precedence; that is, the key given to **dict-get** etc. is regarded as the key to the left map.

```
(make-bimap left-map right-map :key on-conflict)
```

*gauche.dictionary* Creates a new bimap consists of two dictionaries, *left-map* and *right-map*. It is the caller’s responsibility to choose appropriate type of dictionaries; for example, if you want to create a relation between a string and a number, you man want to create it like this:

```
(make-bimap (make-hash-table 'string=?)) ; string -> number
(make-hash-table 'eqv?)) ; number -> string
```

The keyword argument *on-conflict* specifies what will happen when the added entry would conflict the existing entries. The following values are allowed:
This is the default behavior. Duplicate relations are silently removed in order to maintain one-to-one mapping. For example, suppose a bimap between strings and numbers has had ("foo", 1) and ("bar", 2). When you try to put ("bar", 2) with this option, the first two entries are removed. Returns #t.

:supercede

Raises an error when duplicate relations are found.

#f

When duplicate relations are found, does nothing and returns #f.

Note: At this moment, an attempt to add a relation exactly same as the existing one is regarded as a conflict. This limitation may be lifted in future.

**bimap-left** bimap

**bimap-right** bimap

{gauche.dictionary} Returns the left or right map of bimap, respectively. Do not mutate the returned map, or you’ll break the consistency of the bimap.

**bimap-left-get** bimap key :optional default

**bimap-right-get** bimap key :optional default

{gauche.dictionary} Lookup the value corresponding to the key in the left or right map of bimap. If no entry is found for key, default is returned if provided, otherwise an error is raised.

**bimap-left-exists?** bimap key

**bimap-right-exists?** bimap key

{gauche.dictionary} Returns #f if the left or right map of bimap has an entry of the key, #t otherwise.

**bimap-put!** bimap x y :key on-conflict

{gauche.dictionary} Put a relation (x, y) into the bimap. After this, (bimap-left-get x) will return y, and (bimap-right-get y) will return x.

If the bimap already have relations with x and/or y, the conflict is handled according to the value of on-conflict; see make-bimap for the possible values and their meanings. The on-conflict keyword argument can override the bimap’s default setting specified at its creation time.

**bimap-left-delete!** bimap key

**bimap-right-delete!** bimap key

{gauche.dictionary} Deletes an relation with the given left key or right key from bimap. Both left and right maps are modified so that the consistency is maintained. If there’s no relations with given key, these are noop.

**Stacked map**

<stacked-map>

{gauche.dictionary} Stacked map allows you to stack (layer) multiple maps (dictionaries) and treat as if they are a single map. The upper map "shadows" the lower maps. Think of nested scopes, for example.

As a dictionary, it behaves as follows:

- The dict-get operation searches the given key from the top dictionary to the bottom dictionary, returns the first found entry.
- The dict-put! operation always put the entry to the topmost dictionary. If there’s an entry with the same key exists in a lower dictionary, it will be shadowed. If you want to alter the existing entry of non-top dictionary, use stacked-map-entry-update!.
• The dict-delete! operations deletes all the entries with the given key from all the
dictionaries. This is for consistency as a dictionary—you don’t want an entry pop up after
deleting it. If you want to delete only from the first entry in the stack, use stacked-map-
entry-delete!.
• The dict-fold operation traverse all the dictionaries from top to bottom. When there
are duplicate keys, the second and after will be skipped.

make-stacked-map  dic dic2 . . .  [Function]
make-stacked-map  key-comparator dic dic2 . . .  [Function]
Creates a new stacked map with the given dictionaries dic dic2 . . ., where dic is the topmost
dictionary. You can give a comparator key-comparator to be used to compare keys during
traversal with dict-fold. If key-comparator is omitted, the comparator from dic is used.
The key comparator is returned from dict-comparator on the stacked map.

stacked-map-stack  (smap <stacked-map>)  (dic <dictionary>)  [Method]
Creates a new stacked map, adding dic on top of the existing stacked map smap. The key
comparator is inherited from smap.

stacked-map-push!  (smap <stacked-map>)  (dic <dictionary>)  [Method]
Adds dic on top of the dictionaries smap holds.

stacked-map-pop!  (smap <stacked-map>)  [Method]
Remove the topmost dictionary smap holds. An error is signaled if you attempt to pop from
an smap that has no dictionaries.

stacked-map-depth  (smap <stacked-map>)  [Method]
Returns a number of dictionaries smap has.

stacked-map-entry-update!  (smap <stacked-map>)  key proc :optional
fallback  [Method]
Run (dict-update! d key proc fallback) on the first dictionary d that has the given key.
This differs from running dict-update! on smap; if the topmost dictionary doesn’t have an
entry with key but some lower dictionary has, dict-update! takes the existing value and
applies proc to it, then insert the result to the topmost dictionary, while the original entry
remains intact.

stacked-map-entry-delete!  (smap <stacked-map>)  key  [Method]
Deletes the first entry with key. If there’s another entry with key in a lower dictionary, it
will become visible.
This differs from running dict-delete! on smap; it deletes all entries with key, so that
dict-exists? after dict-delete! is guaranteed to return #f.

9.11 gauche.fcntl - Low-level file operations

g挛he.fcntl  [Module]
Provides interface for low-level filesystem operations, including fcntl(2).

sys-fcntl  port-or-fd operation :optional arg  [Function]
{gauche.fcntl} Performs certain operation on the file specified by port-or-fd, which should
be a port object or an integer that specifies a system file descriptor. If it is a port, it must
be associated to the opened file (i.e. port-type returns file, see Section 6.21.3 [Common
port operations], page 260).
The operation is specified by an integer operation. Several variables are defined for valid operation.

**F_GETFD**  Returns flags associated to the file descriptor of port-or-fd. The optional argument arg is not used. The return value is an integer whose definition is system specific, except one flag, FD_CLOEXEC, which indicates the file descriptor should be closed on exec. See the manual entry of fcntl(2) of your system for the details.

**F_SETFD**  Sets the file descriptor flags given as arg to port-or-fd. For example, the portable way of setting FL_CLOEXEC flag is as follows:

```
(sys-fcntl port F_SETFD
 (logior FD_CLOEXEC
 (sys-fcntl port F_GETFD)))
```

**F_GETFL**  Returns flags associated to the open files specified by port-or-fd. The flags includes the following information:

- File access mode. When masked by O_ACCMODE, it’s either one of O_RDONLY, O_WRONLY or O_RDWR.
- File creation options. O_CREAT, O_EXCL and/or O_TRUNC.
- Whether appending is allowed or not, by O_APPEND
- Whether the file is closed automatically on fork.
- Whether I/O is blocking or non-blocking, by O_NONBLOCK.
- Whether it grabs terminal control, by O_NOCTTY.
- Let the system call fail with ELOOP when the pathname is a symbol link.

The system may define system-specific flags.

**F_SETFL**  Sets flags to the open files specified by port-or-fd. Among the flags listed above, only O_NONBLOCK and O_APPEND can be changed.

Note that F_GETFD/F_SETFD concern flags associated to the file descriptor itself, while F_GETFL/F_SETFL concern flags associated to the opened file itself. This makes difference when more than one file descriptor points to the same opened file.

**F_DUPFD**  Creates new file descriptor that points to the same file referred by port-or-fd. An integer must be provided as arg, and that specifies the minimum value of file descriptor to be assigned.

**F_GETLK**  The third argument must be provided and be an instance of <sys-flock> object described below. It searches the lock information specified by arg, and modifies arg accordingly.

**F_SETLK**  The third argument must be provided and be an instance of <sys-flock> object described below. Sets the advisory file lock according to arg. If the lock is successfully obtained, #t is returned. If the other process has the lock conflicting the request, F_SETLK returns #f, while F_SETLKW waits until the lock is available.

**F_GETOWN**  Returns the process id or process group that will receive SIGIO and SIGURG signals for events on the file descriptor. Process group is indicated by a negative value. This flag is only available on the systems that has this feature (BSD and Linux have this).

**F_SETOWN**  Sets the process id or process group that will receive SIGIO and SIGURG signals for events on the file descriptor. Process group is indicated by a negative value.
This flag is only available on the systems that has this feature (BSD and Linux have this). Check out fcntl(2) manpage of your system for the details.

Other value for operation causes an error.

```
<sys-flock>
{gauche.fcntl} A structure represents POSIX advisory record locking. Advisory record locking means the system may not prevents the process from operating on files that it doesn’t have an appropriate lock. All the processes are expected to use fcntl to check locks before it operates on the files that may be shared.

The following slots are defined.

Note that fcntl lock is per-process, per-file. If you try to lock the same file more than once within the same process, it always succeeds. But it’ s not a recursive lock, so the process loses any locks to the file as soon as any of such lock is released, or any of such file is closed. It makes fcntl lock difficult to use in libraries. See with-lock-file (see Section 12.34.6 [Lock files], page 891) for an alternative way to realize inter-process locks.

  type [Instance Variable of <sys-flock>]
  An integer represents lock type. Following variables are predefined for the valid values:

    F_RDLCK    Read locking
    F_WRLCK    Write locking
    F_UNLCK

    To remove a lock by F_SETLK, or to indicate the record is not locked by F_GETLK.

  whence [Instance Variable of <sys-flock>]
  Indicates from where start is measured.

  start [Instance Variable of <sys-flock>]
  The offset of beginning of the locked region.

  len [Instance Variable of <sys-flock>]
  The number of bytes to lock. Zero means “until EOF”.

  pid [Instance Variable of <sys-flock>]
  An integer process id that holding the lock; used only by F_GETLK.

sys-open path flags :optional mode
{gauche.fcntl} A low-level interface corresponds to POSIX open(). Open a file named by path, and returns an integer file descriptor. The file descriptor should be closed with sys-close, or can be turned into a port by open-{input|output}-fd-oprt and to be closed when the port is closed.

This is provided for the code that needs to deal with low-level fd. Unless absolutely necessary, user code should use high-level open-{input|output}-file.

The flags argument is a logical ior of bitmasks O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_CLOEXEC, O_NOCTTY, O_NOFOLLOW, O_NONBLOCK and O_ASYNC. Either one of O_RDONLY, O_WRONLY or O_RDWR must present.

The mode argument specifies the permission bits when a new file is created. The default is #0664.
```
sys-statvfs path
sys-fstatvfs port-or-fd

{gauche.fcntl} Interface to POSIX statvfs and fstatvfs.

Returns information about the filesystem where path, or a file associated to port-or-fd, as <sys-statvfs> instance described below.

These procedures are only defined if the system supports them. You can use the feature identifier gauche.sys.statvfs for the availability (see Section 4.12 [Feature conditional], page 75).

{(cond-expand
   [gauche.sys.statvfs (... code using sys-statvfs ...)]
   [else (... alternative code ...)])

<sys-statvfs> {gauche.fcntl} POSIX struct statvfs. This class is only defined if a feature gauche.sys.statvfs is provided.

bsize [Instance Variable of <sys-statvfs>]
Filesystem block size.

fsize [Instance Variable of <sys-statvfs>]
Fragment size.

blocks [Instance Variable of <sys-statvfs>]
Number of blocks, number of free blocks, and number of free blocks for unprivileged users, in fsize units.

bfree [Instance Variable of <sys-statvfs>]

bavail [Instance Variable of <sys-statvfs>]

files [Instance Variable of <sys-statvfs>]
Number of inodes, number of free inodes, and number of free inodes for unprivileged users.

ffree [Instance Variable of <sys-statvfs>]

favail [Instance Variable of <sys-statvfs>]

fsid [Instance Variable of <sys-statvfs>]
An exact integer filesystem ID.

flag [Instance Variable of <sys-statvfs>]
An exact integer, logical IOR of the bitflags. Portable bitflag values is provided with constants ST_NOSUID a ST_RDONLY.

namemax [Instance Variable of <sys-statvfs>]
Maximum filename length.

ST_NOSUID [Constant]
ST_RDONLY [Constant]
{gauche.fcntl} Bitflag values that can be used in flag slot of <sys-statvfs>. These constants are only defined if a feature gauche.sys.statvfs is provided.

If ST_NOSUID bit is on, suid and sgid bits of the files on this filesystem are ignored by exec(3).

If ST_RDONLY bit is on, the filesystem is mounted read-only.
9.12 gauche.generator - Generators

A generator is merely a procedure with no arguments and works as a source of a series of values. Every time it is called, it yields a value. The EOF value indicates the generator is exhausted. For example, read-char can be seen as a generator that generates characters from the current input port.

It is common practice to abstract the source of values in such a way, so it is useful to define utility procedures that work on the generators. This module provides them.

SRFI-121 (Generators) is a subset of this module. Since gauche.generator predates SRFI-121, we have different names for some procedures; for the compatibility, we provide both names. SRFI-151 (Generators and accumulators) adds some more generator procedures, which is also included (but accumulator procedures are left to SRFI-158. See Section 10.3.12 [R7RS generators], page 631.)

A generator is very lightweight, and handy to implement simple on-demand calculations. However, keep in mind that it is side-effecting construct; you can’t safely backtrack, for example. For more functional on-demand calculation, you can use lazy sequences (see Section 6.18.2 [Lazy sequences], page 241), which is actually built on top of generators.

The typical pattern of using generator is as follows: First you create a source or sources of the values, using one of generator constructors (see Section 9.12.1 [Generator constructors], page 436) or rolling your own one. You may connect generator operators that modifies the stream of generated items as you wish (see Section 9.12.2 [Generator operations], page 440). Eventually you need to extract actual values from the generator to consume; there are utility procedures provided (see Section 9.12.3 [Generator consumers], page 445). Overall, you create a pipeline (or DAG) of generators that works as lazy value-propagation network.

9.12.1 Generator constructors

A generator isn’t a special datatype but just an ordinary procedure, so you can make a generator with lambdas. This module provides some common generator constructors for the convenience.

If you want to use your procedure as a generator, note that a generator can be invoked many times even after it returns EOF once. You have to code it so that once it returns EOF, it keeps returning EOF for the subsequent calls.

The result of generator constructors is merely a procedure, and printing it doesn’t show much.

In the examples in this section we use generator->list to convert the generator to the list. See Section 9.12.3 [Generator consumers], page 445, for the description of generator->list.

null-generator {gauche.generator} An empty generator. Returns just an EOF object when called.

circular-generator arg ... {SRFI-158}gauche.generator Returns an infinite generator that repeats the given arguments.

(generator->list (circular-generator 1 2 3) 10)
⇒ (1 2 3 1 2 3 1 2 3 1)

Note that the above example limits the length of the converted list by 10; otherwise generator->list won’t return.

giota :optional (count +inf.0) (start 0) (step 1) {gauche.generator} Like iota (see Section 6.6.4 [List constructors], page 142), creates a generator of a series of count numbers, starting from start and increased by step.

(generator->list (giota 10 3 2))
⇒ (3 5 7 9 11 13 15 17 19 21)

If both \textit{start} and \textit{step} are exact, the generator yields exact numbers; otherwise it yields inexact numbers.

\[
\text{(generator->list (giota +inf.0 1/2 1/3) 6)}
\Rightarrow (1/2 5/6 7/6 3/2 11/6 13/6)
\]

\[
\text{(generator->list (giota +inf.0 1.0 2.0) 5)}
\Rightarrow (1.0 3.0 5.0 7.0 9.0)
\]

\textit{grange} \texttt{start :optional (end +inf.0) (step 1)} \hfill \text{[Function]}

\{\texttt{gauche.generator}\} Similar to \textit{giota}, creates a generator of a series of numbers. The series begins with \textit{start}, increased by \textit{step}, and continues while the number is below \textit{end}.

\[
\text{(generator->list (grange 3 8))}
\Rightarrow (3 4 5 6 7)
\]

\textit{generate} \texttt{proc} \hfill \text{[Function]}

\{\texttt{gauche.generator}\} Creates a generator from coroutine.

The \texttt{proc} argument is a procedure that takes one argument, \textit{yield}. When called, \textit{generate} immediately returns a generator \textit{G}. When \textit{G} is called, the \texttt{proc} runs until it calls \textit{yield}. Calling \textit{yield} causes to suspend the execution of \texttt{proc} and \textit{G} returns the value passed to \textit{yield}.

Once \texttt{proc} returns, it is the end of the series—\textit{G} returns \texttt{eof} object from then on. The return value of \texttt{proc} is ignored.

The following code creates a generator that produces a series 0, 1, and 2 (effectively the same as \texttt{(giota 3)} and binds it to \textit{g}.

\[
\text{(define g}
\text{ (generate (^
\text{[yield] (let loop ([i 0])}
\text{ (when (< i 3) (yield i) (loop (+ i 1)))))})
\text{)}
\]

\[
\text{(generator->list g)} \Rightarrow (0 1 2)
\]

\textit{list->generator} \texttt{lis :optional start end} \hfill \text{[Function]}

\textit{vector->generator} \texttt{vec :optional start end} \hfill \text{[Function]}

\textit{reverse-vector->generator} \texttt{vec :optional start end} \hfill \text{[Function]}

\textit{string->generator} \texttt{str :optional start end} \hfill \text{[Function]}

\textit{uvector->generator} \texttt{uvec :optional start end} \hfill \text{[Function]}

\textit{bytevector->generator} \texttt{u8vector :optional start end} \hfill \text{[Function]}

\{\texttt{SRFI-158}\}\{\texttt{gauche.generator}\} Returns a generator that yields each item in the given argument. \texttt{reverse-*} procedures runs in reverse order. \texttt{SRFI-121} defines these except \texttt{uvector->generator}, which can take any type of uniform vectors. The \texttt{SRFI-121} version, \texttt{bytevector->generator}, limits the argument to \texttt{u8vector}.

\[
\text{(generator->list (list->generator '1 2 3 4 5))}
\Rightarrow (1 2 3 4 5)
\]

\[
\text{(generator->list (vector->generator '#1 2 3 4 5))}
\Rightarrow (1 2 3 4 5)
\]

\[
\text{(generator->list (reverse-vector->generator '#1 2 3 4 5))}
\Rightarrow (5 4 3 2 1)
\]

\[
\text{(generator->list (string->generator "abcde")})
\Rightarrow (#\text{a} #\text{b} #\text{c} #\text{d} #\text{e})
\]

\[
\text{(generator->list (uvector->generator #'u8(1 2 3 4 5))}
\Rightarrow (1 2 3 4 5)
\]
The generator is exhausted once all items are retrieved; the optional start and end arguments can limit the range the generator walks across; start specifies the left bound and end specifies the right bound.

For forward generators, the first value the generator yields is start-th element, and it ends right before end-th element. For reverse generators, the first value is the item right next to the end-th element, and the last value is the start-th element at the last element, and reverse generators ends at the first element.

\[
\begin{align*}
\text{(generator->list (vector->generator '#(a b c d e) 2))} & \Rightarrow (c d e) \\
\text{(generator->list (vector->generator '#(a b c d e) 2 4))} & \Rightarrow (c d) \\
\text{(generator->list (reverse-vector->generator '#(a b c d e) 2))} & \Rightarrow (d c) \\
\text{(generator->list (reverse-vector->generator '#(a b c d e) 2 4))} & \Rightarrow (b a)
\end{align*}
\]

\[\text{bits->generator } n :\text{optional start end} \quad [\text{Function}]\]
\[\text{reverse-bits->generator } n :\text{optional start end} \quad [\text{Function}]\]

\{gauche.generator\} These procedures take an exact integer and treat it as a sequence of boolean values (0 for false and 1 for true), as \text{bits->list} does (see Section 10.3.22 [R7RS bitwise operations], page 664). \text{Bits->generator} takes bits from LSB, while \text{reverse-bits->generator} takes them from MSB.

\[
\begin{align*}
\text{(generator->list (bits->generator #b10110))} & \Rightarrow (#f #t #t #f #t) \\
\text{(generator->list (reverse-bits->generator #b10110))} & \Rightarrow (#t #f #t #t)
\end{align*}
\]

The optional start and/or end arguments are used to specify the range of bitfield, LSB being 0. Unlike \text{list->generator} etc, start specifies the rightmost position (inclusive) and end specifies the leftmost position (exclusive). It is consistent with other procedures that accesses bit fields in integers (see Section 10.3.22 [R7RS bitwise operations], page 664).

\[
\begin{align*}
\text{(generator->list (bits->generator #x56 0 4))} & \Rightarrow (#f #t #t #f) \; \text{takes bit 0, 1, 2 and 3} \\
\text{(generator->list (bits->generator #x56 4 8))} & \Rightarrow (#t #f #t #f) \; \text{takes bit 4, 5, 6 and 7} \\
\text{(generator->list (reverse-bits->generator #x56 4 8))} & \Rightarrow (#f #t #f #t) \; \text{takes bit 7, 6, 5 and 4}
\end{align*}
\]

Note: SRFI-151’s \text{make-bitwise-generator} is similar to \text{bits->generator}, except that it produces an infinite generator. See Section 10.3.22 [R7RS bitwise operations], page 664.

\[\text{port->sexp-generator } input-port \quad [\text{Function}]\]
\[\text{port->line-generator } input-port \quad [\text{Function}]\]
\[\text{port->char-generator } input-port \quad [\text{Function}]\]
\[\text{port->byte-generator } input-port \quad [\text{Function}]\]

\{gauche.generator\} Returns a generator that reads characters or bytes from the given port, respectively. They’re just \text{(cut read input-port)}, \text{(cut read-line input-port)}, \text{(cut read-char input-port)} and \text{(cut read-byte input-port)}, respectively, but we provide them for completeness.
x->generator obj
{gauche.generator}  A generic version to convert any collection obj to a generator that walks across the obj. Besides, if obj is an input port, port->char-generator is called.

file->generator filename reader . open-args
{gauche.generator}  Opens a file filename, and returns a generator that reads items from the file by a procedure reader, which takes one argument, an input port. The arguments open-args are passed to open-input-file. The file is closed when the generator is exhausted. If a generator is abandoned before being read to the end, then the file is kept open until the generator is garbage-collected. If you want to make sure the file is closed by a certain point of time, you might want to use a reader procedure as a generator within the dynamic extent of with-input-from-file etc.

file->sexp-generator filename . open-args
[Function]
file->char-generator filename . open-args
[Function]
file->line-generator filename . open-args
[Function]
file->byte-generator filename . open-args
[Function]
{gauche.generator}  Returns a generator that reads a series of sexps, characters, lines and bytes from a file filename, respectively. These are versions of file->generator specialized by read, read-char, read-line and read-byte as the reader argument. Like file->generator, open-args are passed to open-input-file (see Section 6.21.4 [File ports], page 264). The file is closed when the generator is exhausted.

gunfold p f g seed :optional tail-gen
{gauche.generator}  A generator constructor similar to unfold (see Section 10.3.1 [R7RS lists], page 593).

P is a predicate that takes a seed value and determines where to stop. F is a procedure that calculates a value from a seed value. G is a procedure that calculates the next seed value from the current seed value. Tail-gen is a procedure that takes the last seed value and returns a generator that generates the tail.

For each call of the resulting generator, p is called with the current seed value. If it returns a true, then we see we’ve done, and tail-gen is called (if it is given) to get a generator for the tail. Otherwise, we apply f on the current seed value to get the value to generate, and use g to update the seed value.

(generator->list (gunfold (^s (> s 5)) (^s (* s 2)) (^s (+ s 1)) 0))
⇒ ’(0 2 4 6 8 10)

giterate f seed
{gauche.generator}  Returns a generator of an infinite sequence of values where the next value is computed by applying f on the current value. The first value generated by giterate is seed itself, while the first one by giterate1 is (f seed).

(generator->list (giterate (pa$ * 2) 1) 10)
⇒ (1 2 4 8 16 32 64 128 256 512)
(generator->list (giterate1 (pa$ * 2) 1) 10)
⇒ (2 4 8 16 32 64 128 256 512 1024)

The reason we have giterate1 is that it’s pretty efficient (up to 10% faster than giterate).

See also literate in gauche.lazy (see Section 9.15 [Lazy sequence utilities], page 451).
SRFI-158 compatible procedures

**generator item ...**

[SRFI-158]{gauche.generator} Returns a generator that generates `item`.

**make-iota-generator count :optional start step**

[SRFI-158]{gauche.generator} Same as `giota`, except that the `count` argument is required.

**make-range-generator start :optional end stop**

[SRFI-158]{gauche.generator} Same as `grange`.

**make-coroutine-generator proc**

[SRFI-158]{gauche.generator} Same as `generate`.

**make-for-each-generator for-each obj**

[SRFI-158]{gauche.generator} Given collection `obj` and walker `for-each`, creates a generator that retrieves one item at a time from the collection. Trivially defined as follows:

```scheme
(define (make-for-each-generator for-each coll)
  (generate (^[yield] (for-each yield coll))))
```

If `obj` is mutated before the returned generator walks all the values, the behavior depends on how the `for-each` procedure handles the situation; it may or may not be safe. In general it’s better to avoid mutation until the generator returns EOF. Once the generator is exhausted, though, it is safe to mutate `obj`.

**make-unfold-generator stop? mapper successor seed**

[SRFI-158]{gauche.generator} This is the same as `gunfold`, except it doesn’t take optional `tail-gen` argument.

### 9.12.2 Generator operations

The following procedures take generators (noted as `gen` and `gen2`) and return a generator. For the convenience, they also accept any collection to `gen` and `gen2` parameters; if a collection is passed where a generator is expected, it is implicitly coerced into a generator.

(NB: This is Gauche’s extension. For portable SRFI-121/SRFI-158 programs, you shouldn’t rely on this behavior; instead, explicitly convert collections to generators.)

**gcons* item ... gen**

[SRFI-158]{gauche.generator} Returns a generator that adds `items` in front of `gen`.

```scheme
(generator->list (gcons* 'a 'b (giota 2)))
⇒ (a b 0 1)
```

**gappend gen ...**

[SRFI-158]{gauche.generator} Returns a generator that yields the items from the first given generator, and once it is exhausted, use the second generator, and so on.

```scheme
(generator->list (gappend (giota 3) (giota 2)))
⇒ (0 1 2 0 1)
```

```scheme
(generator->list (gappend))
⇒ ()
```

**gconcatenate gen**

{gauche.generator} The `gen` argument should generate generators and/or sequences. Returns a generator that yields elements from the first generator/sequence, then the second one, then the third, etc.
It is similar to \((\text{apply } \text{gappend} \ (\text{generator->list} \ \text{gen}))\), except that \text{gconcatenate} can work even \text{gen} generates infinite number of generators.

\[
\begin{align*}
\text{($ generator->list $ gconcatenate} \\
\text{$ list->generator '((\text{giota } 3), (\text{giota } 2)))} \\
\Rightarrow (0 1 2 0 1)
\end{align*}
\]

\text{gflatten gen} \quad \text{[Function]}

\text{SRFI-158}{\{\text{gauche.generator}\}} \text{ The argument \text{gen} is a generator that yields lists. This procedure returns a generator that’s yield each element of the lists one at a time.}

Example: The game Tetris determines the next dropping piece (tetrimino) by the following algorithm: Take a bag of tetriminos with one for each kind (O, I, T, S, Z, L, J), permute it, and draw one by one; once the bag is empty, take another bag and repeat. The algorithm can be implemented by a pipeline of generates as follows. (Tetris is a registered trademark of The Tetris Company).

\[
\begin{align*}
\text{(use gauche.generator)} \\
\text{(use data.random) \ for \ permutations-of}
\end{align*}
\]

\[
\text{(define g}
\begin{align*}
\text{(}$ gflatten $ \text{ permutations-of} \\
\text{(}$ \text{circular-generator '}(0 \ 1 \ T \ S \ Z \ L \ J))\text{))}
\end{align*}
\]

\[
\begin{align*}
\text{($ generator->list $ g 21)} \\
\Rightarrow (L \ 0 \ Z \ T \ J \ S \ I \ J \ L \ Z \ T \ I \ O \ S \ T \ L \ Z \ S \ I \ J \ O)
\end{align*}
\]

Note the subtle difference of this example and the example in \text{gconcatenate} above—\text{gconcatenate} takes a generator of generators, while \text{gflatten} takes a generator of lists.

If we use Haskell-ish type notation, you can see the subtle differences of those similar procedures:

\[
\begin{align*}
gappend & : (\text{Generator a, Generator a, ...}) \rightarrow \text{Generator a} \\
\text{(pa$ apply gappend) & : [(Generator a)] \rightarrow \text{Generator a} \\
gconcatenate & : \text{Generator Generator a} \rightarrow \text{Generator a} \\
gflatten & : \text{Generator} \ [a] \rightarrow \text{Generator a}
\end{align*}
\]

\text{gmerge less-than gen gen2 ...} \quad \text{[Function]}

\text{SRFI-158}{\{\text{gauche.generator}\}} \text{ Creates a generator that yields elements out of input generators, with the order determined by a procedure \text{less-than}. The procedure is called as (\text{less-than a b}) and must return \#t iff the element a must precede the element b.}

Each input generator must yield an ordered elements by itself; otherwise the result won’t be ordered.

If only one generator is given, it is just returned (after coercing the input to a generator). In that case, \text{less-than} won’t be called at all.

\[
\begin{align*}
\text{(generator->list (gmerge < '}(1 \ 3 \ 8) '}(5) '}(2 \ 4)\text{))} \\
\Rightarrow '}(1 \ 2 \ 3 \ 4 \ 5 \ 8)
\end{align*}
\]

\text{gmap proc gen gen2 ...} \quad \text{[Function]}

\text{SRFI-158}{\{\text{gauche.generator}\}} \text{ Returns a generator that yields a value returned by \text{proc} applied on the values from given generators. The returned generator terminates when any of the given generator is exhausted.}

NB: This differs from \text{generator-map} (see Section 6.15.11 [Folding generated values], page 231) which consumes all values at once and returns the results as a list, while \text{gmap} returns a generator immediately without consuming input.
**gmap-accum**  
Function  
{gauche.generator} A generator version of map-accum (see Section 9.6.1 [Mapping over collection], page 405), mapping with states.

The **proc** argument is a procedure that takes as many arguments as the input generators plus one. It is called as (proc v v2 . . . seed) where v, v2, . . . are the values yielded from the input generators, and seed is the current seed value. It must return two values, the yielding value and the next seed.

NB: This is called **gcombine** in SRFI-121.

**gcombine**  
Function  
{SRFI-158}{gauche.generator} An alias of gmap-accum, provided for the compatibility of SRFI-121.

**gfilter**  
Function  
**gremove**  
Function  
{SRFI-158}{gauche.generator} Returns a generator that yields the items from the source generator gen, except those who makes **pred** answers false (**gfilter**) or those who makes **pred** answers a true value (**gremove**)  
(generator->list (gfilter odd? (grange 0)) 6)  
⇒ (1 3 5 7 9 11)  
(generator->list (gremove odd? (grange 0)) 6)  
⇒ (0 2 4 6 8 10)

**gdelete**  
Function  
{SRFI-158}{gauche.generator} Return a generator that yields the items from the source generator gen, except those are the same as **item**. The comparison is done by the procedure passed to =, which defaults to equal?.  
;; Note: This example relies on auto-coercing list to generator.  
;; SRFI-121 requires list->generator for the second argument.  
(generator->list (gdelete 3 '(1 2 3 4 3 2 1)))  
⇒ (1 2 4 2 1)

**gdelete-neighbor-dups**  
Function  
{SRFI-158}{gauche.generator} Returns a generator that yields the items from the source generator gen, but the consecutive items of the same value is omitted. The comparison is done by the procedure passed to =, which defaults to equal?.  
;; Note: This example relies on auto-coercing list to generator.  
;; SRFI-121 requires string->generator for the second argument.  
(generator->list (gdelete-neighbor-dups "mississippi"))  
⇒ (#\m #\i #\s #\i #\s #\i #\p #\i)

**gfilter-map**  
Function  
{SRFI-158}{gauche.generator} Works the same as (**gfilter** values (**gmap** proc gen gen2 . . .)), only slightly efficiently.

**gstate-filter**  
Function  
{SRFI-158}{gauche.generator} This allows stateful filtering of a series. The **proc** argument must be a procedure that takes a value V from the source generator and a seed value. It should return two values, a boolean flag and the next seed value. If it returns true for the boolean flag, the generator yields V. Otherwise, the generator keeps calling **proc**, with updating the seed value, until it sees the true flag value or the source generator is exhausted.
The following example takes a generator of oscillating values and yields only values that are greater than their previous value.

\[
\begin{align*}
\text{(generator->list} & \text{ (gstate-filter (} ^\text{[v s]} \text{ (values (< s v) v)) 0} \\
& \text{(list->generator} \text{ '(1 2 3 2 1 0 1 2 3 1 0 1 2 3)))) \\
\Rightarrow & \text{(1 2 3 1 2 3 1 2 3)}
\end{align*}
\]

\text{gbuffer-filter} \text{ proc seed gen :optional tail-gen}

\section*{Function \text{gbuffer-filter}}

This procedure allows n-to-m mapping between elements in input and output— that is, you can take a look at several input elements to generate one or more output elements.

The procedure \text{proc} receives the next input element and accumulated seed value. It returns two values: A list of output values, and the next seed value. If you need to look at more input to generate output, you can return an empty list as the first value.

If the input reaches the end, \text{tail-gen} is called with the current seed value; it should return a list of last output values. If omitted, the output ends when the output of the last call to \text{proc} is exhausted (the last seed value is discarded).

Suppose you have a text file. Each line contains a command, but if the line ends with backslash, next line is treated as a continuation of the current line. The following code creates a generator that returns \textit{logical} lines, that is, the lines after such line continuations are taken care of.

\[
\begin{align*}
\text{(gbuffer-filter} & \text{ (} ^\text{[v s]} \text{) } \\
& \text{ (if-let1 m (#/\$/ v) (values '() (cons (m 'before) s))} \\
& \text{ (values '((string-concatenate-reverse (cons v s)))) '()))} \\
& \text{ '()} \\
& \text{ (file->line-generator "input-file.txt")} \\
& \text{ '((string-concatenate-reverse s)))))
\end{align*}
\]

\text{gtake} \text{ gen k :optional padding}

\section*{Function \text{gtake}}

\section*{Function \text{gdrop} \text{ gen k}}

\section*{SRFI-158 \{gauche.generator\}}

Returns a generator that takes or drops initial \textit{k} elements from the source generator \textit{gen}.

Those won’t complain if the source generator is exhausted before generating \textit{k} items. By default, the generator returned by \text{gtake} terminates as the source ends, but if you give the optional \textit{padding} argument, then the returned generator does yield \textit{k} items, using the value given to \textit{padding} to fill the rest.

Note: If you pass \textit{padding}, \text{gtake} always returns a generator that generates exactly \textit{k} elements even the input generator is already exhausted—there’s no general way to know whether you’ve reached the end of the input. If you need to take \textit{k} items repeatedly from the input generator, you may want to use \text{gslices} below.

Note for the compatibility: Until 0.9.4, \text{gtake} takes two optional arguments, \textit{fill?} and \textit{padding}. That is consistent with Gauche’s builtin \text{take*}, but incompatible to SRFI-121’s \text{gtake}. We think SRFI-121’s interface is more compact and intuitive, so we renamed the original one to \text{gtake*} (emphasizing the similarity to \text{take*}), and made \text{gtake} compatible to SRFI-121. To ease transition, the current \text{gtake} allows two optional arguments (four in total), in which case we assume the caller wants to call \text{gtake*}: so the code that gives two optional arguments to \text{gtake} would work in both pre-0.9.4 and 0.9.5.

\section*{Function \text{gtake*} \text{ gen k :optional fill? padding}}

\section*{Function \text{gtake*}}

\section*{Section 6.6.5 \[List accessors and...}
modifiers], page 143.) Up to 0.9.4 this version is called \texttt{gtake}. This is provided for the backward compatibility.

\texttt{gtake-while \textit{pred gen}} \hspace{1cm} [Function]
\texttt{gdrop-while \textit{pred gen}} \hspace{1cm} [Function]

[SRFI-158]\{gauche.generator\} The generator version of \texttt{take-while} and \texttt{drop-while} (see Section 6.6.5 [List accessors and modifiers], page 143). The generator returned from \texttt{gtake-while} yields items from the source generator as far as \textit{pred} returns true for each. The generator returned from \texttt{gdrop-while} first reads items from the source generator while \textit{pred} returns true for them, then start yielding items.

\texttt{gslices \textit{gen k} :optional (\textit{fill? #f}) (\textit{padding #f})} \hspace{1cm} [Function]
\{gauche.generator\} The generator version of \texttt{slices} (see Section 6.6.5 [List accessors and modifiers], page 143). This returns a generator, that yields a list of \textit{k} items from the input generator \textit{gen} at a time.

\begin{verbatim}
(generator->list (gslices (giota 7) 3))
⇒ ((0 1 2) (3 4 5) (6))
\end{verbatim}

The \textit{fill?} and \textit{padding} arguments controls how the end of input is handled, just like \texttt{gtake}. When \textit{fill?} is \#f (default), the last item from output generator may not have \textit{k} items if the input is short to fill them, as shown in the above example. If \textit{fill?} is true and the input is short to complete \textit{k} items, \textit{padding} argument is used to fill the rest.

\begin{verbatim}
(generator->list (gslices (giota 6) 3 #t 'x))
⇒ ((0 1 2) (3 4 5))
\end{verbatim}

\begin{verbatim}
(generator->list (gslices (giota 7) 3 #t 'x))
⇒ ((0 1 2) (3 4 5) (6 x x))
\end{verbatim}

\texttt{ggroup \textit{gen k} :optional \textit{padding}} \hspace{1cm} [Function]

[SRFI-158]\{gauche.generator\} Returns a generator lists of \textit{k} elements taken from \textit{gen}. If \textit{padding} is omitted, it works just as \texttt{(gslices \textit{gen} \textit{k})}. If \textit{padding} is given, it works just as \texttt{(gslices \textit{gen} \textit{k} \#t \textit{padding})}.

This is defined in SRFI-158, and more portable than \texttt{gslices}.

\texttt{grxmatch \textit{regexp \textit{gen}}}
\{gauche.generator\} The \textit{gen} argument must be, after coerced, a generator that yields characters.

A generator returned from this procedure tries to match \textit{regexp} from the character sequence generated by \textit{gen}, and once it matches, remember the position after the match and returns \#<\texttt{rxmatch}> object. If no more match is found, the generator is exhausted.

\begin{verbatim}
($ generator->list
 $ gmap rxmatch-substring
 $ grxmatch #/\w+/ "The quick brown fox jumps over the lazy dog.")
⇒ ("The" "quick" "brown" "fox" "jumps" "over" "the" "lazy" "dog")
\end{verbatim}

Note: This procedure is efficient if \textit{gen} is a string, in which case we actually bypass coercing it to a generator. If \textit{gen} is other than a string, the current implementation may need to apply \textit{regexp} as many times as \(O(n^2)\) where \(n\) is the entire length of the character sequence generated by \textit{gen}, although the coefficient is small. This may be improved in future, but be careful using this function on very large input.

Note also that, when \textit{gen} is not a string, \texttt{rxmatch} is applied on some buffered partial input. So \texttt{rxmatch-after} of the returned match does not represents the whole “rest of input” after the match, but merely the rest of strings within the buffer.
Chapter 9: Library modules - Gauche extensions

**gindex** vgen igen

[Function] [SRFI-158] {gauche.generator} Both arguments are generators. The igen generator must yield monotonically increasing series of exact nonnegative integers.

Returns a generator that generates items from vgen indexed by the numbers from igen, exhausted when either source generator is exhausted.

An error is thrown when igen yields a value that doesn’t conform the condition.

```scheme
;; This example takes advantage of Gauche’s auto-coercing
;; list to generator. For portable SRFI-121 programs,
;; you need list->generator for each argument:
(generator->list (gindex '(a b c d e) '(0 2 3)))
⇒ (a c d)
```

**gselect** vgen bgen

[Function] [SRFI-158] {gauche.generator} Both arguments are generators. Creates and returns a generator that yields a value from vgen but only the corresponding value from bgen is true.

The returned generator is exhausted when one of the source generators is exhausted.

```scheme
;; This example takes advantage of Gauche’s auto-coercing
;; list to generator. For portable SRFI-121 programs,
;; you need list->generator for each argument:
(generator->list (gselect '(a b c d e) '(#t #t #f #t #f)))
⇒ (a b d)
```

Combined with a bitgenerator, you can use gselect to extract items using bitmask:

```scheme
(generator->list (gselect '(a b c d e)
(reverse-bits->generator #x1a)))
⇒ (a b d)
```

### 9.12.3 Generator consumers

Some generator consumers are built-in. See Section 6.15.11 [Folding generated values], page 231, for generator-fold, generator-fold-right, generator-for-each, generator-map, and generator-find.

**generator->list** generator :optional k

[Function] [SRFI-158] {gauche.generator} Reads items from generator and returns a list of them (or a reverse list, in case of generator->reverse-list). By default, this reads until the generator is exhausted. If an optional argument k is given, it must be a nonnegative integer, and the list ends either k items are read, or the generator is exhausted.

Be careful not to pass an infinite generator to this without specifying k—this procedure won’t return but hogs all the memory before crash.

**generator->reverse-list** generator :optional k

[Function] [SRFI-158] {gauche.generator} The proc argument must be a procedure that takes as many arguments as the number of given generators.

Returns a list, each of whose element is created by applying proc on each element from given generators gen gen2 ... . The list ends when any of the generator is exhausted.

Note that the list is created eagerly—if all of the generators are infinite, this procedure never returns.

**generator-map->list** proc gen gen2 ...

[Function] [SRFI-158] {gauche.generator} Extracts items from the generator gen up to k items or until it exhausts, and create a fresh vector or string from the extracted items.
When \( k \) is omitted, \( \text{gen} \) is called until it exhausts; note that if \( \text{gen} \) is infinite generator this procedure won’t return.

For \texttt{generator->string}, \( \text{gen} \) must yield a character, or an error is reported.

\[\text{generator->uvector} \; \text{gen :optional} \; k \; \text{class} \]

\[\text{generator->bytevector} \; \text{gen :optional} \; k \]

\[\text{SRFI-158}\{\text{gauche.generator}\} \text{ Extracts items from the generator } \text{gen} \text{ up to } k \text{ items or until it exhausts, and create a fresh uniform vector of class } \text{class} \text{ filled with those items. If } k \text{ is omitted, } \text{gen} \text{ is read until it exhausts.}

If \text{class} is specified, it must be one of the uniform vector classes (see Section 6.13.2 [Uniform vectors], page 198). When omitted, \texttt{<u8vector>} is assumed.

\text{Generator->bytevector} \text{ works like } \text{generator->uvector} \text{ except that the class is fixed to } \texttt{<u8vector>}.\]

The generator must always produce numeric values acceptable to be an element of the specified uvector; otherwise an error is signalled.

\[\text{generator->vector!} \; \text{vector at gen} \]

\[\text{SRFI-158}\{\text{gauche.generator}\} \text{ Fill vector from index } at \text{ with the value yielded from } \text{gen}. \text{It terminates when } \text{gen} \text{ is exhausted or the index reaches at the end of the vector.} \text{ Returns the number of items generated.}

\[
\begin{align*}
 \text{(define v (vector 'a 'b 'c 'd 'e))} \\
 \text{(generator->vector! v 2 (giota))} \\
 \Rightarrow 3 \\
 v \Rightarrow #(a b 0 1 2)
\end{align*}
\]

\[\text{generator->uvector!} \; \text{uvector at gen} \]

\[\text{generator->bytevector!} \; \text{u8vector at gen} \]

\[\{\text{gauche.generator}\} \text{ Like } \text{generator->vector!}, \text{ fill a uniform vector } \text{uvector} \text{ starting from index } at \text{ with elements read from a generator } \text{gen}. \text{It terminates when } \text{gen} \text{ is exhausted or the index reaches at the end of the vector.} \text{ Returns the number of items generated.}

Any type of uvector can be passed to \text{generator->uvector!}, while \text{generator->bytevector!} can only accept \texttt{u8vector}.\]

The generator must always produce numeric values acceptable to be an element of the specified uvector; otherwise an error is signalled.

\[\text{glet* (binding . . .) body body2 . . .} \]

\[\{\text{gauche.generator}\} \text{ This captures a monadic pattern frequently appears in the generator code. It is in a similar spirit of and-let*, but returns as soon as the evaluating expression returns EOF, instead of #f as and-let* does.}\]

The \text{binding} part can be either \texttt{(var expr)} or \texttt{( expr )}. The actual definition will explain this syntax clearly.

\[
\begin{align*}
 \text{(define-syntax glet*)} \\
 \text{(syntax-rules ()} \\
 \text{[(_ () body body2 ...) (begin body body2 ...)]} \\
 \text{[(_ ([ivar gen-exp] more-bindings ...)) \text{. body})} \\
 \text{(let1 var gen-exp} \\
 \text{(if (eof-object? var)} \\
 \text{var} \\
 \text{(glet* (more-bindings ...)) \text{. body}))]} \\
\end{align*}
\]
[(_ ([ gen-expr ] more-bindings ...) . body)
  (let1 var gen-expr
   (if (eof-object? var)
       var
       (glet* (more-bindings ...) . body)))]

```scheme
(glet1 var expr body body2 . . .)
```

This is to `glet*` as `let1` is to `let*`. In other words, it is `(glet* ([var expr]) body body2 . . .)`.

```scheme
do-generator (var gexpr) body . . .
```

This is a generator version of `dolist` and `dotimes` (see Section 4.6 [Binding constructs], page 59).

`Gexpr` is an expression that yields a generator. It is evaluated once. The resulting generator is called repeatedly until it returns EOF. Every time the generator is called, `body . . .` are evaluated in the scope where `var` is bound to the value yielded from the generator.

Like `dolist` and `dotimes`, this macro exists for side-effects. You can write the same thing with `for-each` families, but sometimes this macro makes the imperative code more readable:

```scheme
(do-generator [line (file->line-generator "filename")]
  ;; do some side-effecting stuff with line
)
```

```scheme
generator-any pred gen
```

```scheme
generator-every pred gen
```

Like `any` and `every` (see Section 6.6.6 [Walking over lists], page 147), but works on a generator.

```scheme
generator-count pred gen
```

Returns the number of items in a generator `gen` that satisfies `pred`. As a side effect, `gen` is exhausted.

```scheme
generator-unfold gen unfold arg . . .
```

Apply `unfold` using the values generated from a generator `gen` as seeds. It is equivalent to the following expression:

```scheme
(unfold eof-object? identity (^_ (gen)) (gen) arg . . .)
```

The `unfold` procedure must have the signature `(unfold stop? mapper successor seed arg . . .)`, like `unfold` in `scheme.list` (see Section 10.3.1 [R7RS lists], page 593).

It can be seen as a general method to convert a generator to a sequence, an inverse of `x->generator`.

```scheme
(generator-unfold (x->generator "abc") string-unfold)
⇒ "abc"
```

### 9.13 gauche.hook - Hooks

#### gauche.hook

Provides a hook object, which manages a list of closures to be called at certain time.

This API of hooks are upper-compatible of Guile’s, with the following extensions.

- Based on Gauche’s object system. Most APIs are methods so you can extend the hook features.
- Hook object itself is applicable. You don’t need to use `run-hook`. 

• The method to remove a procedure from a hook is called \texttt{delete-hook!}, for consistency with SRFI-1 and others. \texttt{remove-hook!} is defined as an alias of \texttt{delete-hook!} for compatibility with Guile.

If you’re writing portable code, SRFI-173 provides the basic hook functionality (see Section 11.35 [Hooks (srfi)], page 751).

\begin{verbatim}
<hook> [Class]
{gauche.hook} A hook class, which keeps a list of procedures to be called at once.

The object-apply method is defined on <hook> class, so you can "apply" a hook object as if it were a procedure—which causes all the registered procedure to be invoked.

\[ \textbf{Method}\]
\texttt{add-hook! (hook <hook>) proc :optional (append? #f)}
{gauche.hook} Adds a procedure \texttt{proc} to \texttt{hook}. If \texttt{append?} is given and true, \texttt{proc} is added at the end of the list. Otherwise, \texttt{proc} is added at the front of the list. The \texttt{proc} has to be called with the arity given at \texttt{make-hook}.

\[ \textbf{Method}\]
\texttt{delete-hook! (hook <hook>) proc}
{gauche.hook} Removes \texttt{proc} from the procedure list of \texttt{hook}. \texttt{Remove-hook!} is an alias of \texttt{delete-hook!} just for compatibility with Guile.

\[ \textbf{Method}\]
\texttt{reset-hook! (hook <hook>)}
{gauche.hook} Empties \texttt{hook}'s procedure list.

\[ \textbf{Method}\]
\texttt{hook->list (hook <hook>)}
{gauche.hook} Returns a copy of \texttt{hook}'s procedure list.

\[ \textbf{Method}\]
\texttt{run-hook (hook <hook>) arg . . .}
{gauche.hook} Calls \texttt{hook}'s procedures in order, with arguments \texttt{arg . . .}. The number of arguments must match the arity given at \texttt{make-hook}.
\end{verbatim}

\section{9.14 \texttt{gauche.interactive} - Utilities for interactive session}

\texttt{gauche.interactive} [Module]
Provides useful utilities for the interactive session.
This module is automatically loaded when you run \texttt{gosh} interactively.
This module also sets autoloads for functions defined in \texttt{gauche.reload} module (see Section 9.29 [Reloading modules], page 508), so that those functions can be used by default in interactive development.

\texttt{apropos pattern :optional module} [Macro]
{gauche.interactive} Show a list of defined variables whose name matches \texttt{pattern}. If you give a module or a module name \texttt{module}, only the variables defined in that module are listed. Without \texttt{module}, the variables "visible" from the current module are listed.
pattern may be a symbol or a regexp object. If it is a symbol, the variables whose name
contains the substring that matches the symbol’s name are listed. If it is a regexp object,
the variables whose name matches the regexp are listed.
Some examples:

;;; List variables that contains "string" in their name
(apropos 'string)

;;; Search in srfi.14 module
(apropos 'char 'srfi.14)

d :optional obj
{gauche.interactive} Prints the detail information about a Scheme object obj. The default
method shows obj’s class, and if it has any slots, the list of slot names and their values.
You can specialize this method for customized display. Some built-in types has specialized
methods (see how an integer is described in the example below).
If obj is omitted, the last evaluation result bound to *1 in REPL is used. (see Section 3.2.1
[Working in REPL], page 24)
gosh> (sys-stat "Makefile")
#<sys-stat 0x1e7de60>
gosh> (d)
#<sys-stat 0x1e7de60> is an instance of class <sys-stat>
slots:
  type : regular
  perm : 436
  mode : 33204
  ino : 3242280
  dev : 2097
  rdev : 0
  nlink : 1
  uid : 500
  gid : 500
  size : 19894
  atime : 1435379061
  mtime : 1432954340
  ctime : 1432954340

gosh> (d 1432954340)
1432954340 is an instance of class <integer>
  (#x556925e4, ~ 1.4Gi, 2015-05-30T02:52:20Z as unix-time)

info symbol
{gauche.interactive} Displays an entry of the named function, syntax, module or class
from Gauche’s info document. If an environment variable INFOPATH is defined, this function
searches for the info file from the directories in it. Otherwise, this function guesses info file
location from the gosh’s library directory. If the info file can’t be found, an error is signaled.
So this function doesn’t work if you haven’t installed info file.
If no entry exactly matching with symbol is found, the procedure tries to look for similar
named entries:
gosh> (info 'stirng)
There is no entry for stirng.
Did you mean:
(If you want to search entries using pattern, see info-serach below.)

If the current output port is a tty, the info page is displayed by a paging software. If an environment variable PAGER is defined, it is used as a paging software. Otherwise, this function looks for less and more in this order from the directories in PATH. If none of them is found, or the output port is not a tty, this function just displays the page.

The first invocation of this function in a session takes some time to parse the info file.

NB: When you use less as a pager, make sure you set it to handle utf-8 characters (e.g. setting LESSCHARSET environment variable to UTF-8), or you’ll see some escaped sequences on the screen.

NB: If you invoke gosh within the build tree, using -ftest option, info reads the info files in the build tree if they exist.

info-search regexp
{gauche.interactive} Lists info entries matching regexp. See info above about where the info files are searched.

ed filename-or-procedure :key editor load-after
{gauche.interactive} Invoke an external editor to open the named file, or the file containing the definition of the given procedure (if it can be known). For the latter, it uses source-location procedure to find out the source code location (see Section 6.25.1 [Debugging aid], page 326).

The name of the editor to invoke is determined as follows:
1. The editor keyword argument.
2. The value of the variable *editor* in the user module, if defined. This is handy that you can set this in your .gaucherc.
3. The value of the environment variable GAUCHE_EDITOR.
4. The value of the environment variable EDITOR.

If none of the above is defined or #f, the procedure prompts the user to type in the name of the editor.

Once the editor name is obtained, it is invoked as a subprocess, with the following format:

EDITOR +lineno filename

The lineno is an integer line number, 1 being the first line. The editor is expected to locate the cursor on the specified line.

Once the editor process exits, the procedure checks if the named file is updated. If so, it may load the file, according to the value of the load-after keyword argument. It may take one of the following values:

#t Load the file automatically if it’s updated.

#f Do not load the file.

ask The symbol ask cause the procedure to prompt the user whether it should load the file. This is the default.
9.15 gauche.lazy - Lazy sequence utilities

This module provides utility procedures that yields lazy sequences. For the details of lazy sequences, see Section 6.18.2 [Lazy sequences], page 241.

Since lazy sequences are forced implicitly and indistinguishable from ordinary lists, we don’t need a separate set of procedures for taking lists and lazy sequences; we can use find to search in both ordinary lists and lazy sequences.

However, we do need a separate set of procedures for returning either lists or lazy sequences. For example, lmap can take any kind of sequences, and returns lazy sequence (and calls the procedure on demand).

This distinction is subtle, so I reiterate it. You can use both map and lmap on lazy sequences. If you want the result list at once, use map; it doesn’t have overhead of delayed calculation. If you don’t know you’ll use the entire result, or you know the result will get very large list and don’t want to waste space for an intermediate list, you want to use lmap.

9.15.1 Lazy sequence constructors

You can construct lazy sequences with built-in generator->lseq and lcons. This module provides a few more constructors.

\[ x->lseq \]  \{gauche.lazy\} A convenience function to coerce obj to (possibly lazy) list. If obj is a list, it is returned as it is. If obj is other type of collection, the return value is a lazy sequence that iterates over the collection. If obj is other object, it is returned as it is (you can think of it as a special case of dotted list).

If you try x->lseq in REPL, it looks as if it just converts the input collection to a list.

```
(x->lseq #(a b c)) ⇒ (a b c)
```

But that’s because the lazy sequence is forced by the output routine of the REPL.

\[ lunfold \]  \{gauche.lazy\} A lazy version of unfold (see Section 10.3.1 [R7RS lists], page 593). The arguments p, f and g are procedures, each of which take one argument, the current seed value. The predicate p determines when to stop, f creates each element, and g generates the next seed value. The seed argument gives the initial seed value. If tail-gen is given, it should also be a procedure that takes one argument, the last seed value (that is, the seed value (p seed) returned #f). It must return a (possibly lazy) list, that forms the tail of the resulting sequence.

```
(lunfold ($ = 10 $) ($ * 2 $) ($ + 1 $) 0 (^_ '(end)))
⇒ (0 2 4 6 8 10 12 14 16 18 end)
```

\[ literate \]  \{gauche.lazy\} Creates an infinite sequence of (seed (proc seed) (proc (proc seed)) ...).

The same sequence can be created with (lunfold (^_ #f) identity proc seed), but this one is a lot more efficient.

See also util.stream, which as stream-iterate (see Section 12.90.2 [Stream constructors], page 1034).

```
(take (literate (pa$ + 1) 0) 10)
⇒ (0 1 2 3 4 5 6 7 8 9)
```
coroutine->lseq proc
{gauche.lazy} The proc procedure is called with one argument, yield, which is also a procedure that takes one argument. Whenever yield is called, the value passed to it becomes the next element of resulting lseq. When proc returns, lseq ends.

(coroutine->lseq (^[yield] (dotimes [i 10] (yield (square i)))))
⇒ (0 1 4 9 16 25 36 49 64 81)

See also generate (Section 9.12.1 [Generator constructors], page 436), and coroutine->cseq (Section 12.6 [Concurrent sequences], page 816).

9.15.2 Lazy sequence operations

lseq->list obj
{gauche.lazy} Returns obj, but if it is an lseq, fully computes all values. It is useful when you need to ensure necessary computation is done by certain moment, e.g. you want to ensure all data is read from a port before closing it.

lmap proc seq seq2 ...
{gauche.lazy} Returns a lazy sequence consists of values calculated by applying proc to every first element of seq seq2 ..., every second element of them, etc., until any of the input is exhausted. Application of proc will be delayed as needed.

;; If you use map instead of lmap, this won’t return
(use math.prime)
(take (lmap (pa$ * 2) *primes*) 10)
⇒ (4 6 10 14 22 26 34 38 46 58)

lmap-accum proc seed seq seq2 ...
{gauche.lazy} The procedure proc takes one element each from seq seq2 ..., plus the current seed value. It must return two values, a result value and the next seed value. The result of lmap-accum is a lazy sequence consists of the first values returned by each invocation of proc.

(use math.prime)
(take (lmap-accum (^[p sum] (values sum (+ p sum))) 0 *primes*) 10)
⇒ (0 2 5 10 17 28 41 58 77 100)

This is a lazy version of map-accum (see Section 9.6.1 [Mapping over collection], page 405), but lmap-accum does not return the final seed value. We only know the final seed value when we have the result sequence to the end, so it can’t be calculated lazily.

lappend seq ...
{gauche.lazy} Returns a lazy sequence which is concatenation of seq .... Unlike append, this procedure returns immediately, taking O(1) time. It is useful when you want to append large sequences but may use only a part of the result.

lconcatenate seqs
{gauche.lazy} The seqs argument is a sequence of sequences. Returns a lazy sequence that is a concatenation of all the sequences in seqs.

This differs from (apply lappend seqs), for lconcatenate can handle infinite number of lazy seqs.

lappend-map proc seq1 seq ...
{gauche.lazy} Lazy version of append-map. This differs from a simple composition of lappend and lmap, since (apply lappend (lmap proc seq1 seq ...)) would evaluate the result of lmap to the end before passing it to lappend (it’s because apply need to determine the list of arguments before calling lappend).
It also differs from \((lconcatenate \ (lmap \ proc\ seq1\ seq ...))\) in the subtle way.
Remember that Gauche’s lazy sequence evaluates one element ahead? \(lconcatenate\) does that to the result of \(lmap\). To see the effect, let’s define a procedure with a debug print:

\[
(\text{define} \ (p\ x) \ ?=(\text{list} \ x \ x))
\]

You can see in the following example that \((\text{apply} \ lappend \ (lmap \ ...))\) wouldn’t delay any of application of \(p\):

\[
gosh> \ (\text{car} \ (\text{apply} \ lappend \ (\text{lmap} \ p \ '(1\ 2\ 3)))))
\]

\[
(\text{car} \ (\text{apply} \ lappend \ (\text{lmap} \ p \ '(1\ 2\ 3)))))
\]

\[
?="(\text{standard input})":4:((\text{list} \ x \ x)
\]

\[
?-(1\ 1)
\]

\[
?="(\text{standard input})":4:((\text{list} \ x \ x)
\]

\[
?-(2\ 2)
\]

\[
?="(\text{standard input})":4:((\text{list} \ x \ x)
\]

\[
?-(3\ 3)
\]

\[
1
\]

How about \(lconcatenate\)?
\[
gosh> \ (\text{car} \ (lconcatenate \ (lmap \ p \ '(1\ 2\ 3))))
\]

\[
(\text{car} \ (lconcatenate \ (lmap \ p \ '(1\ 2\ 3))))
\]

\[
?="(\text{standard input})":4:((\text{list} \ x \ x)
\]

\[
?-(1\ 1)
\]

\[
?="(\text{standard input})":4:((\text{list} \ x \ x)
\]

\[
?-(2\ 2)
\]

\[
1
\]

Oops, even though we need only the first element, and the first result of \(lmap\), \((1\ 1)\), provides the second element, too, \(p\) is already applied to the second input.

This is because the intermediate lazy list of the result of \(lmap\) is evaluated “one element ahead”. On the other hand, \(lappend-map\) doesn’t have this problem.

\[
gosh> \ (\text{car} \ (lappend-map \ p \ '(1\ 2\ 3)))
\]

\[
(\text{car} \ (lappend-map \ p \ '(1\ 2\ 3)))
\]

\[
?="(\text{standard input})":4:((\text{list} \ x \ x)
\]

\[
?-(1\ 1)
\]

\[
1
\]

\textbf{linterweave seq ...}

\{\text{gauche.lazy}\} Returns a lazy seq of the first items from \(seq\. . .\), then their second items, and so on. If the length of shortest sequence of \(seqs\) is \(N\), the length of the resulting sequence is \((* N \ number-of-sequences)\). If all of \(seqs\) are infinite, the resulting sequence is also infinite.

\[
(\text{linterweave} \ (\text{lrange} \ 0) \ '(a\ b\ c\ d\ e) \ (\text{circular-list} \ '*))
\]

\[
⇒ \ (0\ a\ *\ 1\ b\ *\ 2\ c\ *\ 3\ d\ *\ 4\ e\ *)
\]

\textbf{lfilter proc seq}

\{\text{gauche.lazy}\} Returns a lazy sequence that consists of non-false values calculated by applying \(proc\) on every elements in \(seq\).

\textbf{lfilter-map proc seq seq2 . . .}

\{\text{gauche.lazy}\} Lazy version of \text{filter-map}.

\textbf{lstate-filter proc seed seq}

\{\text{gauche.lazy}\} Lazy sequence version of \text{gstate-filter} (see Section 9.12.2 [Generator operations], page 440).
ltake seq n :optional fill? padding
ltake-while pred seq
{gauche.lazy} Lazy versions of take* and take-while (see Section 6.6.5 [List accessors and modifiers], page 143). Note that ltake works rather like take* than take, that is, it won’t complain if the input sequence has less than \( n \) elements. Because of the lazy nature of ltake, it can’t know whether input is too short or not before returning the sequence.

There are no ldrop and ldrop-while; you don’t need them. If you apply drop and drop-while on lazy sequence, they return lazy sequence.

lrxmatch rx seq
{gauche.lazy} This is a lazy sequence version of grxmatch (see Section 9.12.2 [Generator operations], page 440).
The seq argument must be a sequence of characters (including ordinary strings). The return value is a lazy sequence of <rxmatch> objects, each representing strings matching to the regular expression \( rx \).
This procedure is convenient to scan character sequences from lazy character sequences, but it may be slow if you’re looking for rarely matching string from very large non-string input. Unless seq is a string, lrxmatch buffers certain length of input, and if matching phrase isn’t found, it extend the buffer and scan again from the beginning, since the match may span from the end of previous chunk to the newly added portion.

lslices seq k :optional fill? padding
{gauche.lazy} Lazy version of slices (see Section 6.6.5 [List accessors and modifiers], page 143).

| (lslices '(a b c d e f) 2) |
| ⇒ ((a b) (c d) (e f)) |

9.15.3 Lazy sequence with positions
Treating input data stream as a lazy sequence is a powerful abstraction; especially, it allows unlimited lookahead with simple list manipulation.
However, you’ll have a difficulty when you want to know the position of the input data within the input stream, e.g. for an error message. Unlike reading from a port, which gives you the current input position, a lazy sequence just looks like a list and unknown amount of data may be prefetched from the real source.

Gauche has special pair objects, called extended pairs, that can carry auxiliary information (see Section 6.6.9 [Extended pairs and pair attributes], page 154). You can create a lazy sequence that carries positional information using the feature.

<sequence-position>
{gauche.lazy} An immutable structure holding positional information. It is returned by lseq-position. The information is queried by the following procedure.

| sequence-position-source seqpos |
| sequence-position-line seqpos |
| sequence-position-column seqpos |
| sequence-position-item-count seqpos |
{gauche.lazy} Query positional information to a <sequence-position> instance seqpos.

Returns the source name (usually the source file name), the line count (starting from 1), the column count (starting from 1), and the item count (number of characters, starting from 0).
The source name may be #f if it is not available.
**Chapter 9: Library modules - Gauche extensions**

**port->char-lseq/position :optional port :key source-name start-line start-column start-item-count**

{gauche.lazy} Like `port->char-lseq`, returns a lazy sequence of characters read from an input port `port`. However, the sequence returned by this procedure has positional info attached, and can be retrieved by `lseq-position`.

The `source-name`, `start-line`, `start-column` and `start-item-count` initializes the positional info before start reading characters. The default values are `(port-name port), 1, 1 and 0, respectively. If you’re reading from a freshly opened port, the default values suffice. Specify these if you’ve already read some data from the port, for example.

**generator->lseq/position char-gen :key source-name start-line start-column start-item-count**

{gauche.lazy} Like `generator->lseq`, returns a lazy sequence of characters generated by `char-gen`. However, the sequence returned by this procedure has positional info attached, and can be retrieved by `lseq-position`.

The `source-name`, `start-line`, `start-column` and `start-item-count` initializes the positional info before start reading characters. The default values are `#f`, 1, 1 and 0, respectively.

**lseq-position seq**

{gauche.lazy} If `seq` is a lazy sequence with positional info attached, retrieve it and returns a `<sequence-position>` instance.

If `seq` doesn’t have positional info, or not even a sequence, `#f` is returned.

### 9.16 `gauche.listener` - Listener

**gauche.listener**

This module provides a convenient way to enable multiple read-eval-print loop (repl) concurrently.

An obvious way to run multiple repls is to use threads; creating as many threads as sessions and calling `read-eval-print-loop` (see Section 6.20 [Eval and repl], page 258) from each thread. Nevertheless, sometimes single threaded implementation is preferred. For instance, you're using a library which is not MT-safe, or your application already uses select/poll-based dispatching mechanism.

To implement repl in the single-threaded selection-base application, usually you register a handler that is called when data is available in the listening port. The handler reads the data and add them into a buffer. Then it examines if the data in the buffer consists a complete expression, and if so, it reads the expression from the buffer, evaluates it, then prints the result to the reporting port. The `<listener>` class in this module provides this handler mechanism, so all you need to do is to register the handler to your dispatching mechanism.

Note: It may also be desirable to buffer the output sometimes, but the current version doesn’t implement it.

**Listener API**

**<listener>**

{gauche.listener} An object that maintains the state of a repl session. It has many external slots to customize its behavior. Those slot values can be set at construction time by using the keyword of the same name as the slot, or can be set by `slot-set!` afterwards. However, most of them should be set before calling `listener-read-handler`.

**input-port**

[Instance Variable of `<listener>`]

Specifies the input port from which the listener gets the input. The default value is the current input port when the object is constructed.
output-port  [Instance Variable of <listener>]
Specifies the output port to which the listener output will go. The default value is the current output port when the object is constructed.

error-port  [Instance Variable of <listener>]
Specifies the output port to which the listener’s error messages will go. The default value is the current error port when the object is constructed.

reader  [Instance Variable of <listener>]
A procedure with no arguments. It should read a Scheme expression from the current input port when called. The default value is system’s read procedure.

evaluator  [Instance Variable of <listener>]
A procedure that takes two arguments, a Scheme expression and an environment specifier. It should evaluate the expression in the given environment and returns zero or more value(s). The default value is system’s eval procedure.

printer  [Instance Variable of <listener>]
A procedure that takes zero or more argument(s) and prints them out to the current output port. The default value is a procedure that prints each value by write, followed by a newline.

prompter  [Instance Variable of <listener>]
A procedure with no arguments. It should prints a prompt to the current output port. The output is flushed by the listener object so this procedure doesn’t need to care about it. The default procedure prints "listener> ".

environment  [Instance Variable of <listener>]
An environment specifier where the expressions will be evaluated. The default value is the value returned by (interaction-environment).

finalizer  [Instance Variable of <listener>]
A thunk that will be called when EOF is read from input-port. During the execution of finalizer, the current input, output and error ports are restored to the ones when listener-read-handler is called.

It can be #f if no such procedure is needed. The default value is #f.

error-handler  [Instance Variable of <listener>]
A procedure that takes one argument, an error exception. It is called when an error occurs during read-eval-print stage, with the same dynamic environment as the error is signaled. The default value is a procedure that simply prints the error exception by report-error.

fatal-handler  [Instance Variable of <listener>]
A procedure that takes one argument, an error exception. It is called when a fatal error occurred (see below for the precise definition). If this handler is called, you should assume you can no longer continue the listener session safely, even write messages to the client. This handler is to log such condition or to clean up the listener. During the execution of fatal-handler, the current input, output and error ports are restored to the ones when listener-read-handler is called.

If fatal-handler returns #f, finalizer is called afterwards. With this, you can implement a common cleanup work in finalizer. If fatal-handler returns a true value, finalizer will not be called.
**listener-read-handler** (listener $<$listener$>$)  
{gauche.listener} Returns a thunk that is to be called when a data is available from input-port of the listener.

The returned thunk (read handler) does the following steps. Note that the first prompt is not printed by this procedure. See **listener-show-prompt** below.

1. Reads available data from input-port and appends it to the listener’s internal buffer.
2. Scans the buffer to see if it has a complete S-expression. If not, returns.
3. Reads the S-expression from the buffer. The read data is removed from the buffer.
4. Evaluates the S-expression, then prints the result to output-port.
5. Prints the prompt by prompter procedure to output-port, then flush output-port.
6. Repeats from 2.

**listener-show-prompt** (listener $<$listener$>$)  
{gauche.listener} Shows a prompt to the listener’s output port, by using listener’s prompter procedure. Usually you want to use this procedure to print the first prompt, for instance, when the client is connected to the listener socket.

**complete-sexp?** str  
{gauche.listener} Returns $\texttt{#t}$ if str contains a complete S-expression. This utility procedure is exported as well, since it might be useful for other purposes.

Note that this procedure only checks syntax of the expressions, and doesn’t rule out erroneous expressions (such as containing invalid character name, unregistered SRFI-10 tag, etc.). This procedure may raise an error if the input contains ‘$<$’ character sequence.

**Error handling**

There are a few error situations the listener handles differently.

- **Fatal error** - An error situation that the listener session can no longer go on safely. You cannot even tell so to the listener client, since the connection to the client may be broken. All you can do is to clean up the listener session (e.g. removes the handler). This case happens in (1) a low-level system error occurs during reading from input-port. (A syntax error of the input isn’t count as fatal, and handled as REPL error described below.), (2) a SIGPIPE signal is raised during writing to output-port, or (3) an unhandled error occurred during executing error-handler.

When this situation happens, the fatal-handler is called if it is given. If fatal-handler returns $\texttt{#f}$, or fatal-handler isn’t given, finalizer is also called.

- **Leaked error** - If an error occurs during executing fatal-handler or finalizer, we don’t have no more safety net. The error is ‘leaked’ outside the listener handler, and should be handled by the user of gauche.listener.

Generally this situation should be considered as a bug of the program; you should make sure to catch foreseeable errors within fatal-handler and finalizer.

- **REPL error** - Other errors are handled by error-handler.

**Listener example**

The following code snippet opens a server socket, and opens a Scheme interactive session when a client is connected. (Note: this code is just for demonstration. Do not run this program on the machine accessible from outside network!)

```scheme
(use gauche.net)
(use gauche.selector)
(use gauche.listener)
```
(define (scheme-server port)
  (let ((selector (make <selector>))
        (server (make-server-socket 'inet port :reuse-addr? #t))
        (cid 0))
    (define (accept-handler sock flag)
      (let* ((client (socket-accept server))
             (id cid)
             (input (socket-input-port client :buffering :none))
             (output (socket-output-port client))
             (finalize (lambda ()
                          (selector-delete! selector input #f #f)
                          (socket-close client)
                          (format #t "client #~a disconnected\n" id))))
       (listener (make <listener>
                     :input-port input
                     :output-port output
                     :error-port output
                     :prompter (lambda () (format #t "client[-a]> " id))
                     :finalizer finalize))
      (handler (listener-read-handler listener)))
    (format #t "client #~a from ~a\n" cid (socket-address client))
    (inc! cid)
    (listener-show-prompt listener)
    (selector-add! selector input (lambda _ (handler)) '(r))))

(selector-add! selector
  (socket-fd server)
  accept-handler
  '(r))

(format #t "scheme server started on port "s\n" port)
(do () (#f) (selector-select selector)))

9.17 gauche.logger - User-level logging

gaque.logger [Module]
  Provides a simple interface to log the program’s activity. The information can be written to the specified file, or to the system logger using syslog(3). When a file is used, syslog-like prefix string is added to each message, which is configurable. It can also takes care of locking of the file (see the description of lock-policy below).

<log-drain> [Class]
  {gauche.logger} Represents the destination of log messages. There’s one implicit global <log-drain> instance, which is used by default. However, you can create as many instances by make method as you want, in case if you want to log to more than one destination.

path [Instance Variable of <log-drain>]
  Designates destination of log output. It can be one of the following values.

  string  Pathname of the log file. The output is written to it.
current-error
#t  The output goes to the current error port.
current-output
The output goes to the current output port.
syslog  The output is sent to the system logger.
ignore  Make log-format does nothing.
#f  The output is turned to a string and returned from log-format.
By default, this slot is #f.

prefix  [Instance Variable of <log-drain>]
Specifies the prefix string that is attached to the beginning of every message. If the message spans to several lines, the prefix is attached to each line. The value of this slot can also be a procedure that takes <log-drain> object and returns a string to be used as the prefix. The procedure is called every time prefix is needed.
When the path slot is a symbol syslog, the value of this slot is ignored. System logger will attach an appropriate prefix.
When the value of the prefix slot is a string, the following character sequences have special meanings and replaced by log-format for appropriate information when written out.

~T  Current time, in the format of "Mmm DD hh:mm:ss" where "Mmm" is an abbreviated month, "DD" is the day of month, "hh", "mm" and "ss" are hours (in 24 hour basis), minutes and seconds, respectively. This format is compatible with system logs.
~Y  Current 4-digit year.
~P  The program name. The default value is the basename of (car (command-line)) (see Section 6.24.2 [Command-line arguments], page 294), but you can change it by the program-name slot described below.
~$  The process id of this program.
~U  The name of the effective user of the process.
~H  The hostname the process is running.
The default value of this slot is "~T ~P[~$]": ". For example, if a string "this is a log message
\line 2\line 3" is given as the message, it produces something like the following log entry.

Sep 1 17:30:23 myprogram[441]: this is a log message
Sep 1 17:30:23 myprogram[441]: line 2
Sep 1 17:30:23 myprogram[441]: line 3

program-name  [Instance Variable of <log-drain>]
Specifies the program name written by ~P directive of the prefix slot.

lock-policy  [Instance Variable of <log-drain>]
Specifies the way the log file should be locked. If the value of this slot is a symbol fcntl, the log file is locked using fcntl() (see Section 9.11 [Low-level file operations], page 432). If the value is a symbol file, the log file is locked by creating auxiliary lock file, whose name is generated by appending ".lock" after the log file path. The logging process needs a write permission to the log file directory. Note that if the process is killed forcibly during writing the log file, a stale lock file may remain. Log-format silently removes the lock file if it is unusually old (currently 10 minutes). If the value is #f, no locking is performed.
The default value is `fcntl`, except MacOSX which doesn’t support `fcntl()`-style locking and thus `file` is default.

The locking isn’t performed if the destination is not a file.

```scheme
[Function]
log-open path :key prefix program-name
{gauche.logger} Sets the destination of the default log message to the path path. It can be a string or a boolean, as described above. You can also set prefix and program name by corresponding keyword arguments. See the `<log-drain>` above for those parameters.

Despite its name, this function doesn’t open the specified file immediately. The file is opened and closed every time `log-format` is called.

[Parameter]
log-default-drain
{gauche.logger} When called with no argument, returns the current default log-drain `log-format` uses when the explicit drain is omitted. It may return `#f` if the default log drain hasn’t been opened by `log-open`.

Calling with new `<log-drain>` object or `#f` alters the default log-drain. You can also use `parameterize` (Section 6.16.1 [Parameters], page 233) to change the log drain temporary.

[Method]
log-format (format <string>) arg ...
{gauche.logger} Formats a log message by format and arg ..., by using format (see Section 6.21.8 [Output], page 275). In the first form, the output goes to the default destination. In the second form, the output goes to the specified drain.

The file is opened and closed every time. You can safely move the log file while your program that touches the log file is running. Also `log-format` acquires a write lock of the log file by `sys-fcntl` (see Section 9.11 [Low-level file operations], page 432).

If the first form of `log-format` is called before `log-open` is called, `log-format` does nothing. It is useful to embed debug stubs in your code; once your code is past the debugging stage, you just comment out `log-open` and the code runs without logging.

9.18 `gauche.mop.instance-pool` - Instance pools

`gauche.mop.instance-pool` Sometimes, you want to track all instances created from a class. This module provies tools to do that.

An instance pool class is a class that keeps the list of instances of itself and its subclasses.

A class that inherits `<instance-pool-mixin>` directly becomes a ‘root’ class of the pool. Instances of the subclass of the root class will be added to the pool.

An application can have multiple root classes. If a class inherits from two or more pooled classes, its instances will be added to all the pools.

The actual implementation of how to manage pools can be customizable by subclassing `<instance-pool-meta>` and overloading methods.

Note that instance pools are global—it keeps all the instances ever created, unless explicitly cleared.
<instance-pool-meta> [Class]
{gauche.mop.instance-pool} A metaclass that adds a class the capability of tracking its instances. By default, instance of itself and its subclasses are tracked.

<instance-pool-mixin> [Class]
{gauche.mop.instance-pool} A mixin class that makes the class that directly inherits this mixin an instance of <instance-pool-meta>. You can use this mixin class, instead of using metaclass explicitly, to make a class a root of the pool.

instance-pool->list class [Generic Function]
{gauche.mop.instance-pool} Returns a list of instances in the pool of class, which should be an instance of <instance-pool-meta>. The list is freshly constructed every time it is called. The order of instances are not specified.

instance-pool-find class pred [Generic Function]
{gauche.mop.instance-pool} Returns an instance of class which satisfies the predicate pred. If no instance satisfies it, #f is returned.

instance-pool-remove! class pred [Generic Function]
{gauche.mop.instance-pool} Remove all instances that satisfies the predicate pred from the pool of class.

instance-pool-fold class kons knil [Generic Function]
{gauche.mop.instance-pool} Calls kons over every instance in the pool of class, with the current accumulated value. The initial value is knil, and each return value of kons is used as the next value. The last return value of kons is returned from the function.

It is functionally equivalent with the following, except that it may be more efficient (without creating intermediate list). Also, the order of instances are not guaranteed to be the same as instance-pool->list returns.

(fold kons knil (instance-pool->list class))

instance-pool-map class proc [Generic Function]
instance-pool-for-each class proc [Generic Function]
{gauche.mop.instance-pool} Apply proc on each instances in the pool of class; instance-pool-map gathers the results into a list and returns it, while instance-pool-for-each discards the result of proc.

The order in which instances are visited is not specified.

9.19 gauche.mop.propagate - Propagating slot access

gauce.mop.propagate [Module]
Provides a metaclass to add :propagated slot allocation option.

When a slot allocation has :propagated, access to the slot is redirected to other object’s slot. It is handy for composite objects to keep external interface simple, for access to the slot of inner objects can be disguised as if it is a slot of the parent object.

An example would work better than explanation. Suppose you have a <rect> class to represent generic rectangular area, and you want to use it when you create a <viewport> class by composition, instead of inheritance. A simple way would be as follows:

(define-class <rect> ()
  ((width :init-keyword :width)
   (height :init-keyword :height)))
(define-class <viewport> ()
  ((dimension :init-form (make <rect>))
   ;; ... other slots ...
))

With this definition, whenever you want to access the viewport's width or height, you have to
go through <rect> object, e.g. (~ viewport' dimension' width). This is not only cumbersome,
but the users of viewport class have to know that how the viewport is composed (it’s not
necessarily a bad thing, but sometimes you may want to hide it).

Using gauche.mop.propagate, you can define slots width and height in <viewport> class
that are proxies of <rect>'s slots.

(use gauche.mop.propagate)

(define-class <rect> ()
  ((width :init-keyword :width)
   (height :init-keyword :height)))

(define-class <viewport> (<propagate-mixin>)
  ((dimension :init-form (make <rect>))
   (width :allocation :propagated :propagate 'dimension
          :init-keyword :width)
   (height :allocation :propagated :propagate 'dimension
           :init-keyword :height)))

With :propagated allocation, the slots are not actually allocated in <viewport> instance,
and accesses to the slots are redirected to the object in the slot specified by :propagate slot
option—in this case, the dimension slot. It is somewhat similar to the virtual slots, but it’s
more convenient for you don’t explicitly write procedures to redirect the access.

Now you can treat width and height as if they are slots of <viewport>. You can even make
them initialize via init-keyword (but you can’t use :init-form or :init-value; if you want
to specify default values, give the default values to the actual object).

  gosh> (define vp (make <viewport> :width 640 :height 480))
  vp
  gosh> (d vp)
  #<<viewport> 0xc5a1e0> is an instance of class <viewport>
  slots:
    dimension : #<<rect> 0xc5a130>
    width : 640
    height : 480
  gosh> (set! (~ vp'width) 800)
  #<undef>
  gosh> (~ vp'width)
  800

Here’s two classes that enables this feature. Usually all you have to do is to inherit
<propagate-mixin> class.

<propagate-meta> [Class]
{gauche.mop.propagate} Adds :propagated slot allocation. The propagated slot has to
have :propagate slot option which specifies the name of the slot that points to an object
that actually holds the value of the slot. If a slot has :propagated slot allocation but does
not have :propagate slot option, an error is signaled.

The :propagate slot option should have a value of either a symbol, or a list of two symbols.
If it is a symbol, it names the slot that contains an object, whose slot with the same name of the propagate slot holds the value.

If it is a list of two symbols as (X Y), then the access to this propagated slot actually works as (slot-ref (slot-ref obj X) Y).

If you want to make a propagated slot initializable by init-keywords, make sure the slot holding the actual object comes before the propagated slots. Slot initialization proceeds in the order of appearance by default, and you want the actual object is created before setting values.

<propagate-mixin>
{gauche.mop.propagate} This is a convenience mixin class. Instead of giving :metaclass <propagate-meta>, you can just inherit this class to make propagated slots available.

9.20 gauche.mop.singleton - Singleton

gauche.mop.singleton
{Module}
Provides a metaclass to define a singleton class.

<singleton-meta>
{gauche.mop.singleton} Creates a singleton class. A singleton class is a class that is guaranteed to create only one instance. The first invocation of make creates the single instance, and further attempt of creation returns the same instance.

(define-class single () () :metaclass <singleton-meta>)

(define a (make single))
(define b (make single))

(eq? a b) ⇒ #t
The slots of the instance are initialized at the first invocation of make. Initargs of make are effective only in the fist invocation, and ignored in the subsequent invocation.

The call of initialization in make is thread-safe.

instance-of (class <singleton-meta>) :rest initargs
{gauche.mop.singleton} This method just calls make with the passed arguments. It is more obvious in the program that you’re dealing with singleton.

<singleton-mixin>
{gauche.mop.singleton} An instance of <singleton-meta>. Instead of specifying <singleton-meta> as the :metaclass argument of define-class, you can inherit this class to give your class the property of singleton.

9.21 gauche.mop.validator - Slot with validator

gauche.mop.validator
{Module}
Provides a metaclass that adds :validator and :observer slot options.

<validator-meta>
{gauche.mop.validator} This metaclass adds a feature that you can specify callbacks that are called before and after the slot value is set. For example, if you want to guarantee that a certain slot always holds a string value, you can make a procedure be called before the slot is modified, either by slot-ref or by a setter method. In the procedure you can either rejects a value except string, or coerce the value to a string.
A *validator* procedure is a callback procedure that is called before the slot value is set. It can be specified by :validator slot option. The procedure takes two values, the instance and the value to be set. Whatever the procedure returns is set to the actual slot value.

A *observer* procedure is a callback procedure that is called after the slot value is set. It can be specified by :observer slot option. The procedure also takes two values, the instance and the new value. Result of the observer procedure is discarded.

See the following example:

```scheme
(define-class <v> ()
  ((a :accessor a-of
       :validator (lambda (obj value) (x->string value)))
   (b :accessor b-of
       :validator (lambda (obj value)
                   (if (integer? value)
                       value
                       (error "integer required for slot b"))))
   :metaclass <validator-meta>))

(define v (make <v>))
(slot-set! v 'a 'foo)
(slot-ref v 'a)  ⇒  "foo"

(set! (a-of v) 1234)
(a-of v)  ⇒  "1234"

(slot-set! v 'b 55)
(slot-ref v 'b)  ⇒  55

(slot-set! v 'b 3.4)  ⇒  error
(set! (b-of v) 3.4)  ⇒  error
```

You can specify default slot value (:init-value etc.) with :validator. In that case, the initialization method of the instance calls the validator with the specified default value, if :init-keyword is not given.

```scheme
(define-class <v> ()
  ((a :initform 'foo :init-keyword :a
       :validator (lambda (obj value) (x->string value)))))

(slot-ref (make <v>) 'a)  ⇒  "foo"
(slot-ref (make <v> :a 555) 'a)  ⇒  "555"
```

It looks similar to the virtual slot, but note that a slot with validator has an actual storage in the instance, while a virtual slot doesn’t.

It is also a good example of customizing how the slots are accessed using the metaobject protocol. This feature is implemented by only a couple of dozen lines of code.

### 9.22 gauche.net - Networking

**gauche.net**  
[Module]  
Provides a set of functions necessary for network communications based on BSD socket interface.
The API is provided in two different levels. Lower level routines reflect traditional BSD socket interface, such as \texttt{bind(2)}. Higher level routines provides more convenient way to create typical connection-oriented server/client sockets.

This module also provides APIs to obtain various information about hostnames, service ports, and protocols.

Gauche can handle IPv6 if it is compiled with the \texttt{--enable-ipv6} configuration option. To check whether IPv6 is enabled or not, you can use \texttt{cond-expand} with \texttt{gauche.net.ipv6} feature identifier after loading \texttt{gauche.net}, as shown below.

\begin{verbatim}
(use gauche.net)
(cond-expand
  (gauche.net.ipv6
   ... code to use ipv6 ...)
  (else
   ... ipv4 only code ...))
\end{verbatim}

See Section 4.12 [Feature conditional], page 75, for the details of \texttt{cond-expand}.

Note: If you want to write a portable program using network, take a look at \texttt{srfi.106} (see Section 11.22 [Basic socket interface], page 727).

### 9.22.1 Socket address

#### Socket address objects

\texttt{<sockaddr>} \hspace{1cm} [Builtin Class]
\{\texttt{gauche.net}\} An abstract base class of socket addresses. Each socket address family is implemented as a subclass of this class.

Although socket addresses are built-in classes, you can use \texttt{make} method to create an instance of a specific socket address family.

\texttt{sockaddr-family \textit{addr}} \hspace{1cm} [Generic Function]
\{\texttt{gauche.net}\} Returns a symbol that indicates the family of the socket address \textit{addr}.

\texttt{sockaddr-name \textit{addr}} \hspace{1cm} [Generic Function]
\{\texttt{gauche.net}\} Returns a string which represents the content of the socket address \textit{addr}.

\texttt{<sockaddr-in>} \hspace{1cm} [Builtin Class]
\{\texttt{gauche.net}\} AF_INET family socket address. To create an instance of this class, use \texttt{make} method as follows:

\begin{verbatim}
(make <sockaddr-in> :host host :port port)
\end{verbatim}

\texttt{host} can be a string, an integer IP address, a \texttt{u8vector} IP address, or one of the keywords :\texttt{any}, :\texttt{broadcast}, :\texttt{none} or :\texttt{loopback}. If it is a string, it is either a host name or a dotted IP notation. Gauche uses \texttt{gethostbyname(3)} to obtain the actual IP address from \texttt{host} parameter. If it is a keyword :\texttt{any}, or :\texttt{broadcast}, the address uses INADDR_ANY, or INADDR_BROADCAST respectively. The keyword :\texttt{loopback} is a synonym to the IPv4 loopback address "127.0.0.1".

\texttt{port} must be a positive integer indicating the port number. See also \texttt{make-sockaddr} below, to create multiple socket addresses on the machine which may have more than one protocol stack.

\texttt{sockaddr-family \textit{addr} <sockaddr-in>)} \hspace{1cm} [Method]
\{\texttt{gauche.net}\} Returns a symbol \texttt{inet}.
sockaddr-name (addr sockaddr-in>)  [Method]
{gauche.net} Returns a string in the form "a.b.c.d:port", where "a.b.c.d" is dotted decimal notion of the IP address and port is the port number.

sockaddr-addr (addr sockaddr-in>)  [Method]
sockaddr-port (addr sockaddr-in>)  [Method]
{gauche.net} Returns the IP address and the port number as an integer, respectively.

sockaddr-un>  [Builtin Class]
{gauche.net} AF_UNIX family socket address. To create an instance of this class, use make method as follows:

(make sockaddr-un :path path)

path must be a string specifying pathname of the socket.

sockaddr-family (addr sockaddr-un>)  [Method]
{gauche.net} Returns a symbol unix.

sockaddr-name (addr sockaddr-un>)  [Method]
{gauche.net} Returns a pathname of the socket address.

sockaddr-in6>  [Builtin Class]
{gauche.net} AF_INET6 family socket address. This is only available if gauche is configured with --enable-ipv6 configure option. The constructor and the slots are the same as sockaddr-in>. See also make-sockaddrs below, to create multiple socket addresses on the machine which may have more than one protocol stack.

make-sockaddrs host port :optional proto  [Function]
{gauche.net} This is a higher-level utility procedure to create all possible inet domain socket addresses that point to host:port of protocol proto. Particularly, if the specified host has both IPv4 and IPv6 addresses, and the running system supports both, then both IPv4 and IPv6 socket addresses are returned. If host has multiple IP addresses, socket addresses are created for each of these IP address. You can make your network application much more portable among different network stack configurations.

Passing #f to host creates the local (server) address. You can also pass a service name (e.g. "http") instead of an integer, to the port argument. The value of proto can be either a symbol tcp or udp, and the default is tcp.

It always returns a list of socket addresses. If the lookup of host is failed, null list is returned.

Address and string conversion

inet-string->address address  [Function]
{gauche.net} Converts string representing of the internet address address to an integer address. If address is parsed successfully, returns two values: the integer address value and the recognized protocol (the constant value 2 (= AF_INET) for IPv4 addresses, and 10 (= AF_INET6) for IPv6 addresses). If address can’t be parsed, #f and #f are returned.

(inet-string->address "192.168.1.1")
⇒ 3232235777 and 2
(inet-string->address "::1")
⇒ 1 and 10
(inet-string->address "::192.168.1.1")
⇒ 3232235777 and 10
(inet-string->address "ffe0::1")
⇒ 340116213421465348979261631549233385 and 10
(inet-string->address "::192.168.1.1")
⇒ 3232235777 and 10
**inet-string->address!**  
*address buf*  
*{gauche.net}* Like *inet-string->address*, but fills the given *u8vector buf* by the parsed address instead of returning it as an integer value. The integer representation of inet addresses is likely to be a bignum, and you can avoid creating bignums with this function. The given *u8vector buf* must be mutable. Returns the protocol on success, or *#f* on failure.

The caller must provide big enough buffer. If *buf* is larger than required, the result is filled from the top of the *u8vector* and the rest of the vector remains intact.

```scheme
(let* ((buf (make-u8vector 16 0))
        (proto (inet-string->address! "192.168.1.1" buf)))
   (list proto buf))
⇒ (2 #u8(192 168 1 1 0 0 0 0 0 0 0 0 0 0 0 0))
```

**inet-address->string**  
*address protocol*  
*{gauche.net}* Converts the given *address* to its string representation of the protocol *protocol*, which can be either 2 (the constant *AF_INET*) or 10 (the constant *AF_INET6*). An integer or a *u8vector* can be used as *address*. If it is a *u8vector*, only the necessary portion of the vector is read; i.e. the vector can be longer than the required length.

```scheme
(inet-address->string 3232235777 AF_INET)  
⇒ "192.168.1.1"
```

```scheme
(inet-address->string '#u8(192 168 1 1) AF_INET)  
⇒ "192.168.1.1"
```

```scheme
(inet-address->string 3232235777 AF_INET6)  
⇒ "::c0a8:101"
```

### 9.22.2 High-level network functions

**<socket>*  
*{gauche.net}* Abstracts a socket, a communication endpoint.

For a connection-oriented socket, you can access the communication channel by two ports associated to the socket, one for input and another for output. *socket-input-port* and *socket-output-port* returns those ports, respectively.

The <socket> class implements <connection> interface. See Section 9.9 [Connection framework], page 426, for the details. The connection-self-address and connection-peer-address methods return a socket address object.

The following three functions are convenient ways to create a connection-oriented socket. Those functions are to provide an easy methods for typical cases, but have less control. If you need more than these functions provide, use low-level interface.

**make-client-socket**  
:optional *address-spec* . . .  
*{gauche.net}* Creates and returns a client socket, connected to the address specified by *address-spec* . . .

```scheme
(make-client-socket 'unix path)  
The client socket is connected to the unix domain server socket of address *path*.
```

```scheme
(make-client-socket 'inet host port)  
The client socket is connected to the inet domain server socket with hostname *host* and port *port*. TCP protocol is assumed. *host* can be either a hostname, or a dotted decimal notation of IPv4 address. If gauche is compiled with –enable-ipv6, IPv6 address notation can also be used. *Port* must be an exact integer specifying a port number, or a string service name (e.g. "http").
```
If gauche is compiled with --enable-ipv6, and the hostname is given, and the hostname has both IPv6 and IPv4 addresses, then IPv6 connection is tried first, and IPv4 is used when IPv6 fails.

(make-client-socket host port)
This works the same as above. This form is for compatibility with STk.

(make-client-socket sockaddr)
If an instance of sockaddr is passed, a socket suitable for sockaddr is opened and then connected to the given address.

This function raises an error if it cannot create a socket, or cannot connect to the specified address.

(make-client-socket 'inet "www.w3.com" 80)
⇒ ; a socket connected to www.w3.com, port 80

(make-client-socket "127.0.0.1" 23)
⇒ ; a socket connected to localhost, port 23

(make-client-socket 'unix "/tmp/.sock"
⇒ ; a socket connected to a unix domain socket "/tmp/.sock"

make-server-socket :optional address-spec ...
{gauche.net} Creates and returns a server socket, listening the address specified by address-spec.

(make-server-socket 'unix path [:backlog num])
The socket is bound to a unix domain socket with a name path. The keyword argument backlog is passed to socket-listen to specify the maximum number of connection request the server can keep before accepting them. The default is 5. If your server is very busy and you see "connection refused" often, you might want to increase it.

(make-server-socket 'inet port [:reuse-addr? flag] [:sock-init proc] [:backlog num])
The socket is bound to an inet domain TCP socket, listening port port, which must be a non-negative exact integer or a string service name (e.g. "http"). If port is zero, the system assigns one of available port numbers. If a keyword argument reuse-addr? is given and true, SO_REUSEADDR option is set to the socket before bound to the port. This allows the process to bind the server socket immediately after other process releases the port.
Alternatively, you can pass a list of positive exact integers to port. In that case, Gauche tries to bind each port in the list until it succeeds.

If keyword argument sock-init is given, it should be a procedure that takes two arguments, a created socket and the socket address. The procedure is called just after the socket is created. It is useful to set some special socket options. The keyword argument backlog is the same as in unix sockets; see the description above.

(make-server-socket port [:reuse-addr? flag] [:sock-init proc][:backlog num])
This is a synonym to the above form (except port must be an integer). This form is backward-compatible with STk's make-server-socket.

(make-server-socket sockaddr [:reuse-addr? flag][:sock-init proc][:backlog num])
This form explicitly specifies the socket address to listen by an instance of sockaddr.
Chapter 9: Library modules - Gauche extensions

(make-server-socket 'inet 8080)
⇒ #<socket (listen "0.0.0.0:8080")>

(make-server-socket 8080)
⇒ #<socket (listen "0.0.0.0:8080")>

(make-server-socket 'inet 0)
⇒ #<socket (listen "0.0.0.0:35628")>

(make-server-socket 'unix "/tmp/.sock")
⇒ #<socket (listen "/tmp/.sock")>

make-server-sockets host port :key reuse-addr? sock-init
{gauche.net} Creates one or more sockets that listen at port on all available network interfaces of host. You can specify a service name (such as "http") to port, as well as an integer port number. Returns a list of opened, bound and listened sockets.

This procedure is particularly useful when the host has multiple protocol stacks, such as IPv4 and IPv6. In that case, this procedure may return a list of IPv4 socket(s) and IPv6 socket(s).

(On some OSes, single socket can listen both IPv4 and IPv6. On such platform, a list of single socket will be returned.)

The meaning of keyword arguments are the same as of make-server-socket.

You can pass 0 to port, just like make-server-socket, to let the system choose an available port number. If pass 0 as port and this procedure returns multiple sockets, it is guaranteed that all the sockets share the same port number.

Several accessors are available on the returned socket object.

socket-address socket
{gauche.net} Returns a socket address associated with socket. If no address has been associated to the socket, #f is returned.

socket-input-port socket :key (buffering :modest)
socket-output-port socket :key (buffering :line)
{gauche.net} Returns an input and output port associated with socket, respectively.

The keyword argument buffering specifies the buffering mode of the port. See Section 6.21.4 [File ports], page 264, for explanation of the buffering mode.

socket-close socket
{gauche.net} Closes socket. All the ports associated to socket are closed as well. Note: as of release 0.7.2, this procedure does not shutdown the connection. It is because socket may be referenced by forked process(es) and you might want to close it without interfering the existing connection. You can call socket-shutdown to shutdown the connection explicitly.

call-with-client-socket socket proc :key input-buffering output-buffering
{gauche.net} socket must be a connected client socket. proc is called with two arguments, an input port that reads from the socket and an output port that writes to the socket. The socket is closed after proc returns or proc raises an error.

The keyword arguments input-buffering and output-buffering are, if given, passed as the buffering keyword arguments of socket-input-port and socket-output-port, respectively.

This is an example of usage of high-level socket functions, a very simple http client.

#!/usr/bin/env gosh
(use gauche.net)

(define (usage)
(display "Usage: swget url\n" (current-error-port))
(exit 1))

;; Returns three values: host, port, and path.
(define (parse-url url)
  (rxmatch-let (rxmatch #/^http://([-A-Za-z\.]+)((\d+))?(/.*)?/ url)
    (#f host #f port path)
    (values host port path)))

(define (get url)
  (receive (host port path) (parse-url url)
    (call-with-client-socket
      (make-client-socket 'inet host (string->number (or port "80")))
      (lambda (in out)
        (format out "GET ~a HTTP/1.0\r\n" path)
        (format out "host: ~a\r\n\r\n" host)
        (flush out)
        (copy-port in (current-output-port)))))))

(define (main args)
  (if (= (length args) 2)
    (get (cadr args))
    (usage))
  0)

\section{Low-level socket interface}

These functions provide APIs similar to the system calls. Those who are familiar to programming with socket APIs will find these functions useful since you can have more detailed control over the sockets.

\noindent \textbf{make-socket} \textit{domain type :optional protocol} \hfill \textit{[Function]}

\{gauche.net\} Returns a socket with specified parameters.

\begin{description}
  \item[PF_UNIX] \hfill \textit{[Constant]}
  \item[PF_INET] \hfill \textit{[Constant]}
  \item[PF_INET6] \hfill \textit{[Constant]}
\end{description}

\{gauche.net\} These constants are bound to the system’s constants \texttt{PF_UNIX}, \texttt{PF_INET} and \texttt{PF_INET6}. You can use those values for \textit{domain} argument of \textit{make-socket}.

(\texttt{PF_INET6} is defined only if the underlying operating system supports IPv6.)

\begin{description}
  \item[AF_UNIX] \hfill \textit{[Constant]}
  \item[AF_INET] \hfill \textit{[Constant]}
  \item[AF_INET6] \hfill \textit{[Constant]}
\end{description}

\{gauche.net\} These constants are bound to \texttt{AF_UNIX}, \texttt{AF_INET} and \texttt{AF_INET6}.

(\texttt{AF_INET6} is defined only if the underlying operating system supports IPv6.)

\begin{description}
  \item[SOCK_STREAM] \hfill \textit{[Constant]}
  \item[SOCK_DGRAM] \hfill \textit{[Constant]}
  \item[SOCK_RAW] \hfill \textit{[Constant]}
\end{description}

\{gauche.net\} These constants are bound to \texttt{SOCK_STREAM}, \texttt{SOCK_DGRAM} and \texttt{SOCK_RAW}, and suitable to pass to the \textit{type} argument of \textit{make-socket}.

\noindent \textbf{socket-fd} \textit{socket} \hfill \textit{[Function]}

\{gauche.net\} Returns an integer system file descriptor of the underlying socket.
**socket-status socket**

*Function*

{gauche.net} Returns an internal status of `socket`, by one of the following symbols.

- **none**: The socket is just created.
- **bound**: The socket is bound to an address by `socket-bind`.
- **listening**: The socket is listening a connection by `socket-listen`.
- **connected**: The socket is connected by `socket-connect` or `socket-accept`.
- **shutdown**: The socket is shutdown by `socket-shutdown`.
- **closed**: The socket is closed by `socket-close`.

**socket-bind socket address**

*Function*

{gauche.net} Binds `socket` to the local network address `address`. It is usually used to associate specific address to the server port. If binding failed, an error is signaled (most likely the address is already in use).

For the inet domain address, you can pass `address` with `port=0`; the system assigns the port number and sets the actual address to the `address` slot of `socket`.

**socket-listen socket backlog**

*Function*

{gauche.net} Listens `socket`. The socket must be already bound to some address. `backlog` specifies maximum number of connection requests to be queued.

**socket-accept socket**

*Function*

{gauche.net} Accepts a connection request coming to `socket`. Returns a new socket that is connected to the remote entity. The original `socket` keeps waiting for further connections. If there’s no connection requests, this call waits for one to come.

You can use `sys-select` to check if there’s a pending connection request.

**socket-connect socket address**

*Function*

{gauche.net} Connects `socket` to the remote address `address`. This is the way for a client socket to connect to the remote entity.

**socket-shutdown socket how**

*Function*

{gauche.net} Shuts down connection of `socket`. If `how` is `SHUT_RD` (or 0), the receive channel of `socket` is disallowed. If `how` is `SHUT_WR` (or 1), the send channel of `socket` is disallowed. If `how` is `SHUT_RDWR` (or 2), both receive and send channels are disallowed. It is an error to call this function on a non-connected socket.

If you shut down the send channel of the socket, the remote peer sees EOF from its receive channel. This is useful if the remote peer expects EOF before sending something back to you.

**socket-getsockname socket**

*Function*

{gauche.net} Returns a `<sockaddr>` instance that is the local address of `socket`.

**socket-getpeername socket**

*Function*

{gauche.net} Returns a `<sockaddr>` instance that is the peer address of `socket`.

**socket-send socket msg :optional flags**

*Function*

{gauche.net} Interfaces to `send(2)` and `sendto(2)`, respectively. Transmits the content of `msg` through `socket`. `msg` can be either a string or a uniform vector; if you send binary packets, uniform vectors are recommended.

Returns the number of octets that are actually sent.

**socket-sendto socket msg to-address :optional flags**

*Function*
When `socket-send` is used, `socket` must already be connected. On the other hand, `socket-sendto` can be used for non-connected socket, and the destination address is specified by a `<sockaddr>` instance `to-address`.

The optional `flags` can be a bitwise OR of the integer constants `MSG_*`. See the system’s manpage of `send(2)` and `sendto(2)` for the details.

```scheme
socket-sendmsg socket msghdr :optional flags
{gauche.net} Sends a packet described by `msghdr` through `socket` using `sendmsg(3)`. The `msghdr` argument must be a string or u8vector, and it must be prepared as a binary representation of `struct msghdr`. A reliable way to build a `msghdr` is to use `socket-buildmsg` described below.

The `flags` argument is the same as `socket-send` and `socket-sendto`.

Returns number of octets sent.

This procedure is not yet supported under the Windows native platform. You can use the feature identifier `gauche.os.windows` to check availability of this procedure (see Section 3.5 [Platform-dependent features], page 34).

```scheme
socket-buildmsg addr iov control flags :optional buf
{gauche.net} Builds a binary representation of `struct msghdr` which is suitable to be given to `socket-sendmsg`. You have to be familiar with `sendmsg(3)` system call to understand this procedure.

The `addr` argument must be an instance of `<sockaddr>` or `#f`. If it is a sockaddr, the `msg_name` field of the `msghdr` is filled with the address.

The `iov` argument must be a vector or `#f`. If it is a vector, each element must be either a string or a u8vector. They are used to fill `msg_iov` field of the `msghdr`. Their contents will be concatenated in the kernel to make a payload.

The `control` argument represents ancillary data, a.k.a. cmsg. It can be `#f` if you don’t need ancillary data. Otherwise, it must be a list in the following form:

```scheme
((level type data) ...)
```

Where `level` and `type` are exact integers, and `data` is either a string or a u8vector. The former two are used to fill cmsg’s `cmsg_level` and `cmsg_type` fields. The `data` is for cmsg’s data (`cmsg_len` is calculated from `data`).

The `flags` argument is used to fill `msg_flags`.

If the `buf` argument is `#f` or omitted, new memories are allocated to construct the `msghdr`. If a mutable u8vector is given to `buf`, `socket-buildmsg` tries to use it to construct the `msghdr` as much as possible; it allocates memory only if `buf` is used up.

Returns the constructed `msghdr` as a u8vector.

This procedure is not yet supported under the Windows native platform. You can use the feature identifier `gauche.os.windows` to check availability of this procedure (see Section 3.5 [Platform-dependent features], page 34).

```scheme
socket-recv! socket buf :optional flags
{gauche.net} Interface to `recv(2)`. Receives a message from `socket`, and stores it into `buf`, which must be a mutable uniform vector. Returns the number of bytes actually written. `socket` must be already connected. If the size of `buf` isn’t enough to store the entire message, the rest may be discarded depending on the type of `socket`.

The optional `flags` can be a bitwise OR of the integer constants `MSG_*`. See the system’s manpage of `recv(2)` for the details.
socket-recvfrom!  socket  buf  addrs  :optional  flags
[gauge.net] Interface to recvfrom(2). Receives a message from socket, which may be
unconnected, and stores it to a mutable uniform vector buf. Like socket-receive, if the size of
buf isn’t enough to store the entire message, the rest may be discarded depending on the
type of socket.

Returns two values; the number of bytes actually written into buf, and an instance of a
subclass of <sys-sockaddr> which shows the sender’s address.

The addrs argument must be a list of instances of socket addresses, optionally its last cdr
being #t (as a special case, if there’s zero addresses to pass, just #t may be given). The
content of each address doesn’t matter; if the protocol family of one of them matches the
sender’s address, the sender’s address is written into the passed sockaddr object. By listing
sockaddrs of possible families, you can count on socket-recvfrom! to allocate no memory
on successful operation. It is useful if you call socket-recvfrom! in a speed-sensitive inner
loop.

If the sender’s address family doesn’t match any of the addresses given to addrs, the behavior
depends on whether the list is terminated by () or #t. If it is terminated by (), (i.e. addrs
is a proper list), the sender’s address is simply discarded and socket-recvfrom! returns
#f as the second value. If the list is terminated by #t, socket-recvfrom! allocates a fresh
sockaddr object and returns it as the second value.

Two simple cases: If you pass () to addrs, the sender’s address is always discarded, which is
useful if socket is connected (that is, you already know your sender’s address). If you pass
#t to addrs, a new socket address object is always allocated for the sender’s address, which
is convenient if you don’t mind memory allocation.

The optional flags can be a bitwise OR of the integer constants MSG_*. See the system’s
manpage of recvfrom(2) for the details.

socket-receive  socket  bytes  :optional  flags
socket-recvfrom  socket  bytes  :optional  flags
[gauge.net] Like socket-receive! and socket-recvfrom!, but these returns the received
message as a (possibly incomplete) string, up to bytes size. Additionally, socket-recvfrom
always allocates a socket address object for the sender’s address.

The use of these procedures are discouraged, since they often returns incomplete strings
for binary messages. Using strings for binary data creates many pitfalls. Uniform vectors
(especially u8vectors) are for binary data. (The reason these procedures return strings is
merely historical.)

MSG_CTRUNC  [Variable]
MSG_DONTROUTE  [Variable]
MSG_EOR  [Variable]
MSG_OOB  [Variable]
MSG_PEEK  [Variable]
MSG_TRUNC  [Variable]
MSG_WAITALL  [Variable]
[gauge.net] Pre-defined integer constants to be used as flags values for socket-send,
socket-sendto, socket-receive and socket-recvfrom. Some of these constants may not be
defined if the underlying operating system doesn’t provide them.

Further control over sockets and protocol layers is possible by getsockopt/setsockopt interface,
as described below.
socket-setsockopt socket level option value
socket-getsockopt socket level option rsize

{gauche.net} These are the interface to setsockopt() and getsockopt() calls. The interface is a bit clumsy, in order to allow full access to those low-level calls.

socket must be a non-closed socket object. level and option is an exact integer to specify the level of protocol stack and the option you want to deal with. There are several variables pre-bound to system constants listed below.

To set the socket option, you can pass either an exact integer or a string to value. If it is an integer, the value is passed to setsockopt(2) as C\int value. If it is a string, the byte sequence is passed as is. The required type of value depends on the option, and Gauche can’t know if the value you passed is expected by setsockopt(2); it is your responsibility to pass the correct values.

To get the socket option, you need to tell the maximum length of expected result by rsize parameter, for Gauche doesn’t know the amount of data each option returns. socket-getsockopt returns the option value as a byte string. If you know the option value is an integer, you can pass 0 to rsize; in that case socket-getsockopt returns the value as an exact integer.

Note about the name: I tempted to name these function socket-\{set|get\}opt or socket-\{set|get\}-option, but I rather took the naming consistency. Hence duplicated "sock"s.

The following predefined variables are provided. Note that some of them are not available on all platforms. See manpages socket(7), tcp(7) or ip(7) of your system to find out exact specification of those values.

For “level” argument:

SOL_SOCKET
SOL_TCP
SOL_IP

{gauche.net} These variables are bound to SOL_SOCKET, SOL_TCP and SOL_IP, respectively.

For “option” argument:

SO_KEEPALIVE
{gauche.net} Expects integer value. If it is not zero, enables sending of keep-alive messages on connection-oriented sockets.

SO_OOBINLINE
{gauche.net} Expects integer value. If it is not zero, out-of-band data is directly placed into the receive data stream. Otherwise out-of-band data is only passed when the MSG_OOB flag is set during receiving.

SO_REUSEADDR
{gauche.net} Expects integer value. If it is not zero, socket-bind allows to reuse local addresses, unless an active listening socket bound to the address.

SO_TYPE
{gauche.net} Gets the socket type as an integer (like sock_stream). Can be only used with socket-getsockopt.

SO_BROADCAST
{gauche.net} Expects integer value. If it is not zero, datagram sockets are allowed to send/receive broadcast packets.
SO_PRIORITY                  [Variable]
{gauche.net}                 Expects integer value, specifying the protocol-defined priority for all packets
                             to be sent on this socket.

SO_ERROR                     [Variable]
{gauche.net}                 Gets and clears the pending socket error as an integer. Can be only used with
                             socket-getsockopt.

inet-checksum packet size    [Function]
{gauche.net}                 Calculates one’s complement of Internet Checksum (RFC1071) of the packet,
                             which must be given as a uniform vector. First size bytes of packet are used for calculation.
                             Returned value is in network byte order (big-endian). It is an error if size is greater than
                             the size of packet.
                             Note: The used algorithm assumes packet is not too big (< 64K).

9.22.4 Netdb interface

<sys-hostent>                 [Builtin Class]
{gauche.net}                 A class of objects for network hosts. Corresponding to struct hostent in C.
                             The following slots are available read-only.

name                        [Instance Variable of <sys-hostent>]
                             The formal name of the host (string).

aliases                     [Instance Variable of <sys-hostent>]
                             A list of alias names of the host (list of strings).

addresses                   [Instance Variable of <sys-hostent>]
                             A list of addresses (list of strings). Only ipv4 address is supported currently. Each address
                             is represented by dotted decimal notation.

sys-gethostbyname name      [Function]
{gauche.net}                 Looks up a host named name. If found, returns a <sys-hostent> object.
                             Otherwise, returns #f.
                             (let ((host (sys-gethostbyname "www.w3c.org")))
                              (list (slot-ref host 'name)
                                     (slot-ref host 'aliases)
                                     (slot-ref host 'addresses)))
                             ⇒ ("www.w3.org" ("www.w3c.org") ("18.29.1.34" "18.29.1.35"))

sys-gethostbyaddr addr proto [Function]
{gauche.net}                 Looks up a host that has an address addr of protocol proto. addr is a natural
                             string representation of the address; for ipv4, it is a dotted decimal notation. proto is
                             a protocol number; only AF_INET is supported currently. If the host is found, returns a
                             <sys-hostent> object. Otherwise, returns #f.
                             (let ((host (sys-gethostbyaddr "127.0.0.1" AF_INET)))
                              (list (slot-ref host 'name)
                                     (slot-ref host 'aliases)
                                     (slot-ref host 'addresses)))
                             ⇒ ("localhost" ("localhost.localdomain") ("127.0.0.1"))

<sys-servent>                [Builtin Class]
{gauche.net}                 An entry of the network service database. Corresponding to struct servent
                             in C. The following slots are available read-only.
name
The formal name of the service (string).

aliases
A list of alias names of the service (list of strings).

port
A port number registered for this service (exact integer).

proto
A protocol name for this service (string).

sys-getservbyname name proto
{gauche.net} Looks up the network service database with a service name name and a protocol proto. Both name and proto must be a string. If a service is found, an instance of <sys-servent> is returned. Otherwise, #f is returned.

(let ((serv (sys-getservbyname "http" "tcp")))
  (list (slot-ref serv 'name)
        (slot-ref serv 'aliases)
        (slot-ref serv 'port)
        (slot-ref serv 'proto)))
⇒ ("http" () 80 "tcp")

sys-getservbyport port proto
{gauche.net} Looks up the network service database with a service port port and a protocol proto. port must be an exact integer, and proto must be a string. If a service is found, an instance of <sys-servent> is returned. Otherwise, #f is returned.

(let ((serv (sys-getservbyport 6000 "tcp")))
  (list (slot-ref serv 'name)
        (slot-ref serv 'aliases)
        (slot-ref serv 'port)
        (slot-ref serv 'proto)))
⇒ ("x-server" () 6000 "tcp")

<sys-protoent>
{gauche.net} An entry of the protocol database. Corresponds to struct protoent in C. The following slots are available read-only.

name
The formal name of the protocol (string).

aliases
A list of alias names of the protocol (list of strings).

proto
A protocol number (exact integer).

sys-getprotobynum name
{gauche.net} Looks up the network protocol database with a name name, which must be a string. If a protocol is found, an instance of <sys-protoent> is returned. Otherwise, #f is returned.

(let ((proto (sys-getprotobynum "icmp")))
  (list (slot-ref proto 'name)
        (slot-ref proto 'aliases)
        (slot-ref proto 'proto)))
⇒ ("icmp" ("ICMP") 1)
Chapter 9: Library modules - Gauche extensions

sys-getprotobynumber number
{gauche.net} Looks up the network protocol database with a protocol number number, which must be an exact integer. If a protocol is found, an instance of <sys-protoent> is returned. Otherwise, #f is returned.

(let ((proto (sys-getprotobynumber 17)))
  (list (slot-ref proto 'name)
        (slot-ref proto 'aliases)
        (slot-ref proto 'proto)))
⇒ ("udp" ("UDP") 17)

<sys-addrinfo>
{gauche.net} The new interface to keep address information. Corresponds to struct addrinfo in C. This is only available if gauche is configured with –enable-ipv6 option. The following slots are provided.

flags
family
socktype
protocol
addrlen
addr

sys-getaddrinfo nodename servname hints
{gauche.net} Returns a list of <sys-addrinfo> instances from the given nodename, servname and hints. This is only available if gauche is compiled with –enable-ipv6 option.

sys-ntohs integer
sys-ntohl integer
sys-htons integer
sys-htonl integer
{gauche.net} Utility functions to convert 16bit (s) or 32bit (l) integers between network byte order (n) and host byte order (h).

Scheme API to the netdb interface calls those byte order conversion functions internally, so you don’t usually need them so much as in C programs. However, it may be useful when you’re constructing or analyzing binary packets. See also Section 12.2 [Packing binary data], page 811, to handle binary data.

9.23 gauche.package - Package metainformation

gauche.package
Gauche manages extra libraries and extension modules as packages.

Each package source tree has package.scm on top directory, which contains define-gauche-package form that provides metainformation about the package—the package name, version, author, dependencies, etc.

When the package is installed, the standard installation process copies that information, with additional information such as the version of Gauche used to build the package, into .packages subdirectory of the library installation path, with the name PACKAGENAME.gpd, where PACKAGENAME is the name of the package.

We collectively call package.scm and *.gpd as package description file.

This module provides utility procedures to read and write package description files, and search installed *.gpd files.
package.scm

A package file, package.scm, must contain one package definition in the following form. It is not evaluated; it is read as a literal data.

**define-gauche-package** name key-value-list ...  

[Package definition]

Defines a package name. It is followed by a keyword-value list. The following keywords are recognized.

- **version**  The version of the package, in a string, e.g. "1.0".
- **description**  The description of the package, in a string. The first line (up to the first newline character) should be the one-line summary of the package.
- **require**  A list of requirements. Each requirement is (<package-name> <version-spec>), where <package-name> is a string package name, and <version-spec> determines the acceptable versions of the package. See Section 9.39 [Comparing version numbers], page 570, for the details of <version-spec>.
- **providing-modules**  A list of module names in symbols, that this package provides.
- **authors**  A list of name and contact info of the authors of this package.
- **maintainers**  A list of name and contact info of the maintainers of this package, if they differ from the authors.
- **licenses**  A list of licenses.
- **homepage**  A homepage URL of the package, if any.
- **repository**  A repository URL of the package, if any.

**configure script and *.gpd file generation**

If you use configure script based on gauche.configure (see Section 9.8 [Generating build files], page 413), a Gauche package description file (*.gpd) is created with it. The gpd file contains information from package.scm, plus the information on the installation platform gathered by configure.

The generated gpd file is installed with the package itself, and will be used by gauche-package command, as well as retrieved by the following utility APIs.

**Utility procedures**

```scheme
<gauche-package-description>

{gauche.package} An object to handle package descriptions programatically.

- **name** [Instance Variable of <gauche-package-description>]  
  The name of the package, a string

- **version** [Instance Variable of <gauche-package-description>]  
  The version of the package, a string, e.g. "1.0".

- **description** [Instance Variable of <gauche-package-description>]  
  The description of the package. Up to the first newline character may be used as a short summary.
```
require  [Instance Variable of <gauche-package-description>]
A list of (package-name version-spec), to specify the other packages this package requires.

providing-modules  [Instance Variable of <gauche-package-description>]
A list of module names (symbols) that this package provides.

authors  [Instance Variable of <gauche-package-description>]
A list of author’s name and contact info.

maintainers  [Instance Variable of <gauche-package-description>]
A list of maintainer’s name and contact info, if it differs from authors.

licenses  [Instance Variable of <gauche-package-description>]
A list of licenses.

repository  [Instance Variable of <gauche-package-description>]
An URL to the repository of this package.

homepage  [Instance Variable of <gauche-package-description>]
An URL to the homepage of this package.

gauche-version  [Instance Variable of <gauche-package-description>]
Gauche version with which this module is installed.

configure  [Instance Variable of <gauche-package-description>]
A command-line string keeping how configure script was run to build and install this package.

path->gauche-package-description filename  [Function]
{gauche.package} The named file must be a gpd file. This reads the file and returns an instance of <gauche-package-description>.
An error is thrown if filename can’t be read, or doesn’t have a proper define-gauche-package form.

write-gauche-package-description description :optional oport  [Function]
{gauche.package} Write out the content of <gauche-package-description> instance, description as a define-gauche-package form, to an output port oport. If oport is omitted, current output port is used.

make-gauche-package-description name :key version description
require maintainers authors licenses homepage repository gauche-version configure providing-modules
{gauche.package} Creates and returns a new instance of <gauche-package-description>, the slots of which is initialized with the given keyword arguments.

gauche-package-description-paths :key all-versions  [Function]
{gauche.package} Gather all the gpd file paths installed in the standard location on the system. By default, it collects packages installed in *load-path*. If you give a true value to all-version, it attempts to collect packages installed for other versions of Gauche as well.

find-gauche-package-description name :key all-versions  [Function]
{gauche.package} Try to find a package description of name installed in the standard location, and returns an instance of <gauche-package-description> if found. Returns #f if the named package isn’t found.
By default, it collects packages installed in *load-path*. If you give a true value to all-version, it attempts to collect packages installed for other versions of Gauche as well.
9.24 gauche.parameter - Parameters (legacy)

gauche.parameter  
This module is only provided for the backward compatibility. We strongly discourage using this module. Parameters are now built-in, so you no longer need to use this module for basic parameter functionalities. However, they have slightly different semantics from the legacy parameters Gauche had. The parameters provided in this module is backward compatible, so that you can run legacy code without changes.

When you use this module, those two identifiers are rebound to the backward-compatibility API.

make-parameter  
This becomes an alias of make-legacy-parameter.

parameterize  
This becomes an alias of parameterize/dynwind

See Section 6.16 [Parameters and dynamic states], page 232, for the details of the difference.

parameter-observer-add!  
{gauche.parameter} Deprecated. Adds proc to "observer" procedures of a parameter p. Observer procedures are called either (1) just before a new value is set to the parameter, or (2) just after the new value is set to the parameter. In case of (1), a filter procedure is already applied before a callback is called. In either case, observer procedures are called with two arguments, the old value and the new value. The return value(s) of observer procedures are discarded.

The optional when argument must be either a symbol before or after, to specify whether proc should be called before or after the value is changed. If omitted, after is assumed.

The optional where argument must be either a symbol append or prepend, to specify whether proc should be prepended or appended to the existing observer procedure list. If omitted, append is assumed.

Note: Although the parameter value itself is thread-local, the observer list is shared by all threads.

parameter-observer-delete!  
{gauche.parameter} Deprecated. Deletes proc from observer procedure list of a parameter p. If proc is not in the list, nothing happens. You can give either a symbol before or after to when argument to specify from which list proc should be deleted. If when argument is omitted, proc is deleted from both lists.

parameter-pre-observers p  
{gauche.parameter} Deprecated. Returns a hook object (see Section 9.13 [Hooks], page 447) that keeps "before" or "after" observers, respectively.

Note: Although the parameter value itself is thread-local, these hook objects are shared by all threads.

9.25 gauche.parseopt - Parsing command-line options

gauche.parseopt  
This module defines a convenient way to parse command-line options. The interface is hinted by Perl, and conveniently handles long-format options with multiple option arguments.
Actually, you have a few choices to parse command-line options in Gauche. SRFI-37 (see Section 11.10 [A program argument processor], page 709) provides functional interface to parse POSIX/GNU compatible argument syntax. SLIB has getopt-compatible utility. Required features may differ from application to application, so choose whichever fits your requirement.

**High-level API**

**let-args** args (bind-spec ... [. rest]) body ...  
{gauche.parseopt} This macro captures the most common pattern of argument processing. It takes a list of arguments, args, and scans it to find Unix-style command-line options and binds their values to local variables according to bind-spec, then executes body ....

Let’s look at a simple example first, which gives you a good idea of what this form does. (See the “Examples” section below for more examples).

```
(define (main args)
  (let-args (cdr args)
    ((verbose "v\|verbose")
     (outfile "o\|outfile=s")
     (debug-level "d\|debug-level=i" 0)
     (help "h\|help" => (cut show-help (car args)))
     . restargs
    )
    ....))

(define (show-help programe)
  ...
)
```

The local variable verbose will be bound to #t if a command-line argument -v or --verbose is given, and to #f otherwise. The variable output is specified to take one option argument; if the command-line arguments are given like -o out.txt, outfile receives "out.txt". The debug-level one is similar, but the option argument is coerced to an integer, and also it has default value 0 when the option isn’t given. The help clause invokes an action rather than merely binding the value.

(Note: Currently let-args does not distinguish so-called short and long options, e.g. -v and --v have the same effect, so as -verbose and --verbose. In future we may add an option to make it compatible with getopt_long(3).)

The final restargs variable after the dot receives a list of non-optional command-line arguments.

Let’s look at bind-spec in detail. It must be one of the following forms.

1. (var option-spec)
2. (var option-spec default)
3. (var option-spec => callback)
4. (var option-spec default => callback)
5. (else => handler)
6. (else formals body ...)

A list of command-line arguments passed to args are parsed according to option-specs. If the corresponding option is given, a variable var is bound to a value as follows:

(a) If the bind-spec is 1. or 2., then
   (a1) If option-spec doesn’t require an argument, then #t:
   (a2) If option-spec requires one argument, then the value of
the argument:

(a3) If option-spec requires more than one argument, the list of the values of the arguments.
(b) If the bind-spec is 3. or 4., then callback is called with the value(s) of arguments, and its return value.

We’ll explain the details of option-spec later.

As a special case, var can be #f, in which case the value is ignored. It is only useful for side effects in callback.

If the corresponding option is not given in args, var is bound to default if it is given, or #f otherwise.

The last bind-spec may be the form 5 or 6. in which case the clause is selected when no other option-spec matches a given command-line option. In the form 5, handler will be called with three arguments; the given option, a list of remaining command-line arguments, and a continuation procedure. The handler is supposed to handle the given option, and it may call the continuation procedure with the remaining arguments to continue processing, or it may return a list of arguments which will be treated as non-optional command-line arguments. The form 6 is a shorthand notion of (else => (lambda形式als body ...)).

The bind-spec list can be an improper list, whose last cdr is a symbol. In which case, a list of the rest of the command-line arguments is bound to the variable named by the symbol.

Note that the default, callback, and forms in else clause is evaluated outside of the scope of binding of vars (as the name let-args implies).

Unlike typical getopt or getopt_long implementation in C, let-args does not permute the given command-line arguments. It stops parsing when it encounters a non-option argument (argument without starting with a minus sign).

If the parser encounters an argument with only two minus signs ‘--’, it stops argument parsing and returns a list of arguments after ‘--’.

After all the bindings is done, body ... are evaluated. Body may began with internal define forms.

Option spec

option-spec is a string that specifies the name of the option and how the option takes the arguments. An alphanumeric characters, underscore, plus and minus sign is allowed for option’s names, except that minus sign can’t be the first character, i.e. the valid option name matches a regexp #/[\w+][-\w+]*/.

If the option takes argument(s), it can be specified by attaching equal character and a character (or characters) that represents the type of the argument(s) after the name. The option can take more than one arguments. The following characters are recognized as a type specifier of the option’s argument.

s  String.

n  Number.

f  Real number (coerced to flonum).

i  Exact integer.

e  S-expression.

y  Symbol (argument is converted by string->symbol).

Let’s see some examples of option-spec:

"name"  Specifies option name, that doesn’t take any argument.
"name=s" Option name takes one argument, and it is passed as a string.

"name=i" Option name takes one argument, and it is passed as an exact integer.

"name=ss" Option name takes two arguments, both string.

"name=iii" Option name takes three integer arguments.

"name=sf" Option name takes two arguments, the first is a string and the second is a number.

If the option has alternative names, they can be concatenated by "|". For example, an option spec "h|help" will match both "h" and "help".

In the command line, the option may appear with preceding single or double minus signs. The option’s argument may be combined by the option itself with an equal sign. For example, all the following command line arguments match an option spec "prefix=s".

- prefix /home/shiro
  -prefix /home/shiro
  -- prefix /home/shiro
  --prefix=/home/shiro

If the option consists of a single letter and takes arguments, the first argument can follow immediately after the option letter without a whitespace. If you have an option spec "I=s", all the following command line arguments are recognized:

- I/foo
- I /foo
- I=foo
- --I
- --I=foo

If there’s an ambiguity between a long option and a single-letter option plus an argument, the long option takes precedence. If you have an option spec "long" and "l=s", command line arguments -long is recognized as the former, not -l option with arguments long.

Error handling

<parseopt-error> [Condition Type]

{gauche.parseopt} When let-args encounters an argument that cannot be processed as specified by option specs, an error of condition type <parseopt-error> is raised. The cases include when a mandatory option argument is missing, or when an option argument has a wrong type.

(let-args '("-a" "foo") ((a "a=i")) ; option a requires integer
(list a))
⇒ parseopt-error

Note that this condition is about parsing the given args. If an invalid option-spec is given, an ordinary error is thrown.

Examples

This example is taken from gauche-install script. The mode option takes numbers in octal, so it uses the callback procedure to convert it. See also the else clause how to handle unrecognized option.

(let-args (cdr args)
((#f "c") ;; ignore for historical reason
  (mkdir "d\directory")
  (mode "m\mode=s" #o755 => (cut string->number <> 8))
  (owner "o\owner=s")
  (group "g\group=s")
  (srcdir "S\srcdir=s")
  (target "T\target=s")
  (utarget "U\uninstall=s")
  (shebang "shebang=s")
  (verb "v")
  (dry "n\dry-run")
  (#f "h\help" => usage)
  (else (opt . _) (print "Unknown option : " opt) (usage))
  . args)
(...)

The next example is a small test program to show the usage of else clause. It gathers all options into the variable r, except that when it sees -c it stops argument processing and binds the rest of the arguments to restargs.

(use gauche.parseopt)

(define (main args)
  (let1 r ()
    (let-args (cdr args)
      ((else (opt rest cont)
         (cond [[(equal? opt "c") rest]
                [else (push! r opt) (cont rest)])
        . restargs)
        (print "options: " (reverse r))
        (print "restargs: " restargs)
        0)))))

Sample session of the above script (suppose it is saved as example).

$ ./example -a -b -c -d -e foo
options: (a b)
restargs: (-d -e foo)
$ ./example -a -b -d -e foo
options: (a b d e)
restargs: (foo)

Low-level API

The followings are lower-level API used to build let-args macro.

parse-options args (option-clause ...)
  {gauche.parseopt} args is an expression that contains a list of command-line arguments. This macro scans the command-line options (an argument that begins with ‘-’) and processes it as specified in option-clauses, then returns the remaining arguments. Each option-clause is consisted by a pair of option-spec and its action.

If a given command-line option matches one of option-spec, then the associated action is evaluated. An action can be one of the following forms.

bind-spec body ...

bind-spec is a proper or dotted list of variables like lambda-list. The option’s arguments are bound to bind-spec, then then body ... is evaluated.
=> proc  If a command-line option matches option-spec, calls a procedure proc with a list of the option's arguments.

If a symbol else is at the position of option-spec, the clause is selected when no other option clause matches a given command-line option. Three “arguments” are associated to the clause; the unmatched option, the rest of arguments, and a procedure that represents the option parser.

make-option-parser (option-clause ...)
{gauche.parseopt} This is a lower-level interface. option-clauses are the same as parse-options. This macro returns a procedure that can be used later to parse the command line options.

The returned procedure takes one required argument and one optional argument. The required argument is a list of strings, for given command-line arguments. The optional argument may be a procedure that takes more than three arguments, and if given, the procedure is used as if it is the body of else option clause.

9.26 gauche.partcont - Partial continuations

gauche.partcont
{Module}
Gauche internally supports partial continuations (a.k.a. delimited continuations) natively. This module exposes the feature for general use.

Note: Partial continuations use two operators, reset and shift. Those names are introduced in the original papers, and stuck in the programming world. Unfortunately those names are too generic as library function names. We thought giving them more descriptive names, but decided to keep them after all; when you talk about partial continuations you can’t get away from those names. If these names conflict to other names in your program, you can use :prefix import specifier (see Section 4.13.4 [Using modules], page 81), for example as follows:

;; Add prefix pc: to the 'reset' and 'shift' operators.
(use gauche.partcont :prefix pc:)

reset expr ...
{gauche.partcont} Saves the current continuation, and executes expr ... with a null continuation or empty continuation. The shift operator captures the continuation from the shift expression to this null continuation.

Note on implicit delimited continuations: There’s an occasion Gauche effectively calls reset internally: When C routine calls back to Scheme in non-CPS manner. (If you know C API, it is Scm_EvalRec(), Scm_ApplyRec*, Scm_Eval() and Scm_Apply() family of functions.) The callers of such routines expect the result is returned at most once, which won’t work well with Scheme’s continuations that have unlimited extent. Such calls create delimited continuations implicitly.

For example, the main routine of gosh calls Scheme REPL by Scm_Eval(), which means the entire REPL is effectively surrounded by a reset. So, if you call shift without corresponding reset, the continuation of shift becomes the continuation of the entire REPL—which is to exit from gosh. This may be surprising if you don’t know about the implicit delimited continuation.

Other places the implicit delimited continuations are created are the handlers virtual ports (see Section 9.40 [Virtual ports], page 572), object-apply methods called from write and display, and GUI callbacks such as the one registered by glut-display-func (See the document of Gauche-gl for the details), to name a few.
In general you don’t need to worry about it too much, since most built-in and extension routines written in C calls back Scheme in CPS manner, and works with both full and delimited continuations.

**shift var expr ...**

*gauche.partcont* {gauche.partcont} Packages the continuation of this expression until the current null continuation marked by the most recent **reset** into a procedure, binds the procedure to var, then executes expr ... with the continuation saved by the most recent **reset**.

That is, after executing expr ..., the value is passed to the expression waiting for the value of the most recent **reset**. When a partial continuation bound to var is executed, its argument is passed to the continuation waiting for the value of this **shift**. When the execution of the partial continuation reaches its end, it returns from the expression waiting for the value of invocation of var.

**call/pc proc**

*gauche.partcont* This is a wrapper of **shift**. (shift k expr ...) is equivalent to (call/pc (lambda (k) expr ...)). Sometimes this similarity of **call/cc** comes handy.

Well, ... I bet you feel like your brain is twisted hard unless you are one of those rare breed from the land of continuation. Let me break down what’s happening here informally and intuitively.

Suppose a procedure A calls an expression B. If A expects a return value from B and continue processing, we split the part after returning from B into a separate chunk A', then we can think of the whole control flow as this straight chain:

A -> B -> A'  
A -> B is a procedure call and B -> A' is a return, but we all know procedure call and return is intrinsically the same thing, right?

Procedure B may call another procedure C, and so on. So when you look at an execution of particular piece of code, you can think of a chain of control flow like this:

... -> A -> B -> C -> .... -> C' -> B' -> A' -> ...

The magic procedure **call/cc** picks the head of the chain following its execution (marked as * in the figure below), and passes it to the given procedure (denoted k in the figure below). So, whenever k is invoked, the control goes through the chain from *.

... -> A -> B -> (call/cc -> (lambda (k) ... ) ) -> B' -> A' -> ...

One difficulty with **call/cc** is that the extracted chain is only one-ended—we don’t know what is chained to the right. In fact, what will come after that depends on the whole program; it’s outside of local control. This global attribute of **call/cc** makes it difficult to deal with.

The **reset** primitive cuts this chain of continuation. The original chain of continuation (the x-end in the following figure) is saved somewhere, and the continuation of **reset** itself becomes open-ended (the o-end in the following figure).

... -> A -> B -> (reset ... ) -> o

x -> B' -> A' -> ...

A rule: If control reaches to the o-end, we pick the x-end most recently saved. Because of this, **reset** alone doesn’t show any difference in the program behavior.

Now what happens if we insert **shift** inside **reset**? This is a superficial view of inserting **shift** into somewhere down the chain of reset:

... -> (reset -> X -> Y -> (shift k ... ) -> Y' -> X' ) -> o
What actually happens is as follows.

1. \texttt{shift} packages \textit{the rest of the chain of work} until the end of \texttt{reset}, and bind it to the variable \( k \).

2. The continuation of \texttt{shift} becomes a null continuation as well, so after \texttt{shift} returns, the control skips the rest of operations until the corresponding \texttt{reset}.

\[ \ldots \rightarrow (\texttt{reset} \rightarrow X \rightarrow Y \rightarrow (\texttt{shift} \ k \ldots ) \rightarrow o \rightarrow o \rightarrow Y' \rightarrow X' ) \rightarrow o \]

In other words, when you consider the \texttt{reset} form as one chunk of task, then \texttt{shift} in it \textit{stashes away} the rest of the task and immediately returns from the task.

Let's see an example. The \texttt{walker} argument in the following example is a procedure that takes a procedure and some kind of collection, and applies the procedure to the each element in the collection. We ignore the return value of \texttt{walker}.

\begin{verbatim}
(define (inv walker)
  (lambda (coll)
    (define (continue)
      (reset (walker (lambda (e) (shift k (set! continue k) e)) coll)
        (eof-object)))
    (lambda () (continue))))
\end{verbatim}

A typical example of \texttt{walker} is \texttt{for-each}, which takes a list and applies the procedure to each element of the list. If we pass \texttt{for-each} to \texttt{inv}, we get a procedure that is inverted \textit{inside-out}.

What does that mean? See the following session:

\begin{verbatim}
gosh> (define inv-for-each (inv for-each))
inv-for-each
inv-for-each
(gosh> (define iter (inv-for-each '(1 2 3)))
iter
(gosh> (iter))
1
(gosh> (iter))
2
(gosh> (iter))
3
(gosh> (iter))
#<eof>

When you pass a list to \texttt{inv-for-each}, you get an iterator that returns each element in the list for each call. That's because every time \texttt{iter} is called, \texttt{shift} in \texttt{inv} stashes away the task of \textit{walking the rest of the collection} and set it to \texttt{continue}, then returns the current element \( e \).

\texttt{walker} doesn't need to work just on list. The following function \texttt{for-each-leaf} traverses a tree and apply \texttt{f} on each non-pair element.

\begin{verbatim}
(define (for-each-leaf f tree)
  (match tree
    [(x . y) (for-each-leaf f x) (for-each-leaf f y)]
    [x (f x)]))
\end{verbatim}

And you can inverse it just like \texttt{for-each}.

\begin{verbatim}
gosh> (define iter2 ((inv for-each-leaf) '((1 . 2) . (3 . 4))))
iter2
(gosh> (iter2))
1
(gosh> (iter2))
\end{verbatim}
The `util.combinations` module (see Section 12.81 [Combination library], page 1015) provides a procedure that calls a given procedure with every permutation of the given collection. If you pass it to `inv`, you get a procedure that returns every permutation each time.

```scheme
(define next ((inv permutations-for-each) '(a b c)))
(next)
(next)
(next)
(next)
(next)
(next)
(next)
(next)
(next)
#<eof>
```

**9.27 gauche.process - High-level process interface**

This module provides a higher-level API of process control, implemented on top of low-level system calls. This module also provides “process ports”, a convenient way to send/receive information to/from subprocesses.

**9.27.1 Running subprocess**

```scheme
do-process cmd/args :key redirects input output error fork directory host sigmask on-abnormal-exit
run-process cmd/args :key redirects input output error fork directory host sigmask wait
```

{gauche.process} Runs a command with arguments given to `cmd/args` in a subprocess. The `cmd/args` argument must be a list, whose car specifies the command name and whose cdr is the command-line arguments.

If the command name contains a slash, it is taken as the pathname of the executable. Otherwise the named command is searched from the directories in the PATH environment variable.

Each element in `cmd/args` are converted to a string by `x->string`, for the convenience.

Do-process always waits the subprocess to terminate, and returns `#t` if it exits successfully (i.e. with zero exit status). If the subprocess terminates abnormally, the behavior is controled by `on-abnormal-exit` keyword argument:
- If it is #f (default), #f is returned.
- If it is :error, a <process-abnormal-exit> error is thrown.
- If it is :exit-code, and the subprocess terminates with non-zero exit status, the integer exit code (the integer value passed to subprocess’s exit()) is returned. Note that if the subprocess is terminated with a signal, a <process-abnormal-exit> is still thrown.

Do-process! is like do-process except that it raises <process-abnormal-exit> error when the process exists with non-zero status. It’s the same behavior as giving :error to the on-abnormal-exit keyword argument of do-process. It is often more convenient to let the commands fail in shell-script type tasks.

Run-process can run the subprocess concurrently by default, that is, it returns immediately. The return value is a <process> object, which can be used to track the status of the subprocess (see Section 9.27.3 [Process object], page 494).

For example, the following expression runs ls -al.

```scheme
(do-process '(ls -al))
```

You see the output of ls -al, then it returns #t, unless the execution of ls command fails with some reason.

Since do-process returns the success or failure of the command by a boolean value, you can use and and or to combine commands pretty much the same way as shell’s && and || operators.

```scheme
;; shell: make && make -s check
(and (do-process '(make))
 (do-process '(make -s check)))
```

```scheme
;; shell: mv x.tmp x.c || rm -f x.tmp
(or (do-process '(mv x.tmp x.c))
 (do-process '(rm -f x.tmp)))
```

If you use run-process instead, you’ll get <process> object without waiting ls -al to finish. If you run the following expression on REPL, you’ll likely to see the return value before output of ls.

```scheme
(run-process '(ls -al))
```

You can keep the returned <process> object and call process-wait on it to wait for its termination. See Section 9.27.3 [Process object], page 494, for the details of process-wait.

```scheme
(let1 p (run-process '(ls -al))
 ... do some other work ... 
 (process-wait p))
```

You can tell run-process to wait for the subprocess to exit; in that case, run-process calls process-wait internally. It is useful if you want to examine the exit status of the subprocess, rather than just caring its success/failure as do-process does.

Note that -i is read as an imaginary number, so be careful to pass -i as a command-line argument; you should use a string, or write |-i| to make it a symbol.

```scheme
(run-process '(ls "-i"))
```

Note: An alternative way to run external process is sys-system, which takes a command line as a single string (see Section 6.24.10 [Process management], page 318). The string is passed to the shell to be interpreted, so you can include redirections, or can pipe several commands. It would be handy for quick throwaway scripts.

On the other hand, with sys-system, if you want to change command parameters at runtime, you need to worry about properly escape them (actually we have one to do the job in gauche.process; see shell-escape-string below); you need to be aware that /bin/sh,
used by sys-system via system(3) call, may differ among platforms and be careful not to rely on specific features on certain systems. As a rule of thumb, keep sys-system for really simple tasks with a constant command line, and use run-process and do-process for all other stuff.

Note: Old version of this procedure took arguments differently, like (run-process "ls" "-al" :wait #t), which was compatible to STk. This is still supported but deprecated.

Large number of keyword arguments can be passed to do-process and run-process to control execution of the child process. We describe them by categories.

**Synchronization**

**wait flag**

This can only be given to run-process. If flag is true, run-process waits until the subprocess terminates, by calling process-wait internally. Otherwise the subprocess runs asynchronously and run-process returns immediately, which is the default behavior.

Note that if the subprocess is running asynchronously, it is the caller’s responsibility to call process-wait at a certain timing to collect its exit status.

```scheme
;;; This returns after wget terminates.
(define p (run-process '(wget http://practical-scheme.net/) :wait #t))

;;; Check the exit status
(let1 st (process-exit-status p)
  (cond [(sys-wait-exited? st)
         (print "wget exited with status " (sys-wait-exit-status st))]
        [(sys-wait-signaled? st)
         (print "wget interrupted by signal " (sys-wait-termsig st))]
        [else
         (print "wget terminated with unknown status " st)]))
```

**on-abnormal-exit how**

This can only be given to do-process. If how is #f, which is the default, do-process returns #f when the subprocess exits abnormally (i.e. with nonzero exit status). If how is :error, it raises an error in such a case.

**fork flag**

If flag is true, do-process and run-process forks to run the subprocess, which is the default behavior. If flag is false, do-process and run-process directly calls sys-exec, so it never returns.

**I/O redirection**

**redirects (iospec ...)**

Specifies how to redirect child process’s I/Os. Each iospec can be one of the followings, where fd, fd0, and fd1 are nonnegative integers referring to the file descriptor of the child process.

(Note: If you just want to run a command and get its output as a string take a look at process-output->string (see Section 9.27.4 [Process ports], page 498). If you want to pipe multiple commands together, see Section 9.27.2 [Running process pipeline], page 494.)

```scheme
(< fd source)
```

source can be a string, a symbol, a keyword :null, an integer, or an input port.

If it is a string, it names a file opened for read and the child process can reads the content of the file from fd. An error is signaled if the file does not exist or cannot open for read.
If it is a symbol, an unidirectional pipe is created, whose reader end is connected to the child’s \(fd\), and whose writer end is available as an output port returned from `(process-input process source)`.

If it is `:null`, the child’s \(fd\) is connected to the null device.

If it is an integer, it should specify a parent’s file descriptor opened for read. The child sees the duped file descriptor as \(fd\).

If it is an input port, the underlying file descriptor is duped into child’s \(fd\). It is an error to pass an input port without associated file descriptor (See `port-file-number` in Section 6.21.3 [Common port operations], page 260).

\[
<<<<fd \text{value})
\]

Feeds `value` or `obj` to the input file descriptor \(fd\) of the child process.

With `<<`, `value` must be either a string or a uniform vector (see Section 6.13.2 [Uniform vectors], page 198). It is sent to the child process as is. Using a uniform vector is good to pass binary content.

With `<<<`, `obj` can be any Scheme object, and the result of `(write-to-string obj)` is sent to the child process.

\[
<\& fd0 fd1)
\]

Makes child process’s file descriptor \(fd0\) refer to the same input as its file descriptor \(fd1\). Note the difference from `<; (< 3 0) makes the parent’s stdin (file descriptor 0) be read by the child’s file descriptor 3, while `<& 3 0) makes the child’s file descriptor 3 refer to the same input as child’s stdin (which may be redirected to a file or something else by another `iospec`).

See the note below on the order of processing `<&`.

\[
> fd sink)
\]

\[
>>(fd sink)
\]

`sink` must be either a string, a symbol, a keyword `:null`, an integer or a file output port.

If it is a string, it names a file. The output of the child to the file descriptor \(fd\) is written to the file. If the named file already exists, `>` first truncates its content, while `>>` appends to the existing content.

For other arguments, `>` and `>>` works the same.

If `sink` is a symbol, an unidirectional pipe is created whose writer end is connected to the child’s \(fd\), and whose reader end is available as an input port returned by `(process-output process sink)`.

If `sink` is `:null`, child’s \(fd\) is connected to the system’s null device.

If `sink` is an integer, it must specify a parent’s file descriptor opened for output.

The child sees the duped file descriptor as \(fd\).

If `sink` is an output port, the underlying file descriptor is duped into \(fd\) in the child process.

\[
>& fd0 fd1)
\]

Makes child process’s file descriptor \(fd0\) refer to the same output as its file descriptor \(fd1\). Note the difference from `>; (> 2 1) makes the child’s stderr go to parent’s stdout, while (>\& 2 1) makes the child’s stderr go to the same output as child’s stdout (which may be redirected by another `iospec`).

```scheme
;; Read both child’s stdout and stderr
(let1 p (run-process ’(command arg))
```
(begin0 (port->string (process-output p 'out))
 (process-wait p)))

Note: You can’t use the same name (symbol) more than once for the pipe of source or sink. For example, the following code signals an error:

    (run-process '(command) :redirects '((> 1 out) (> 2 out))) ; error!

You can use >& to “merge” the output to one sink, or <& to “split” the input from one source, instead:

    (run-process '(command) :redirects '((> 1 out) (> 2 1)))

It is allowed to give the same file name more than once, just like the Unix shell. However, note that the file is opened individually for each file descriptor, so simply writing to them may not produce desired result (for regular files, most likely that one output would overwrite another).

Note: I/O redirections are processed at once, unlike the way unix shell does. For example, both of the following expression works the same way, that is, they redirect both stdout and stderr to a file out.

    (run-process '(command arg) :redirects '((>& 2 1) (> 1 "out")))
    (run-process '(command arg) :redirects '((> 1 "out") (> 2 1)))

Most unix shells process redirections in order, so the following two command line works differently: The first one redirects child’s stderr to the current stdout, which is the same as the parent’s stdout, then redirects child’s stdout to a file out. So the error messages appear in the parent’s stdout. The second one first redirects the child’s stdout to a file out, so at the time of processing 2>&1, the child’s stderr also goes to the file.

    $ command arg 2>&1 1>out
    $ command arg 1>out 2>&1

You can say do-process and run-process always works like the latter, regardless of the order in redirects argument.

If you want to redirect child’s stderr to parent’s stdout, you can use > like the following:

    (run-process '(command arg) :redirects '((> 2 1) (> 1 "out")))

Redirects child’s standard i/o. source and sink may be either a string, one of keywords :null, :pipe, or merge, an integer file descriptor, or a symbol.

These are really shorthand notations of the redirects argument:

    :input x  ≡ :redirects '((< 0 x))
    :output x  ≡ :redirects '((> 1 x))
    :error x  ≡ :redirects '((> 2 x))

The keyword :pipe as source or sink is supported just for the backward compatibility. They work as if a symbol stdin, stdout or stderr is given, respectively:

    :input :pipe  ≡ :redirects '((< 0 stdin))
    :output :pipe  ≡ :redirects '((> 1 stdout))
    :error :pipe  ≡ :redirects '((> 2 stderr))

That is, a pipe is created and its one end is connected to the child process’s stdin, and the other end is available by calling (process-input process), (process-output process) or (process-error process). (That is because process-input and process-output uses
stdin and stdout respectively when name argument is omitted, and (process-error p) is equivalent to (process-output p 'stderr).

The keyword :merge can only be used for :error keyword argument, and it is a shorthand notation of :redirects '((>& 2 1)), that is, merge the child process’s stderr into child process’s stdout.

See the description of redirects above for the meanings of the argument values.

**Execution environment**

**directory directory**  
[Subprocess argument]  
If a string is given to directory, the process starts with directory as its working directory. If directory is #f, this argument is ignored. An error is signaled if directory is other type of objects, or it is a string but is not a name of a existing directory.

When host keyword argument is also given, this argument specifies the working directory of the remote process.

Note: do-process and run-process check the validity of directory, but actual chdir(2) is done just before exec(2), and it is possible that chdir fails in spite of previous checks. At the moment when chdir fails, there’s no reliable way to raise an exception to the caller, so it writes out an error message to standard error port and exits. A robust program may take this case into account.

**sigmask mask**  
[Subprocess argument]  
Mask must be either an instance of <sys-sigset>, a list of integers, or #f. If an instance of <sys-sigset> is given, the signal mask of executed process is set to it. A list of integers are treated as a list of signals to mask. It is important to set an appropriate mask if you call run-process from multithreaded application. See the description of sys-exec (Section 6.24.10 [Process management], page 318) for the details.

If the host keyword argument is specified, this argument merely sets the signal mask of the local process (ssh).

**detached flag**  
[Subprocess argument]  
When a true value is given, the new process is detached from the parent’s process group and belongs to its own group. It is useful when you run a daemon process. See sys-fork-and-exec (see Section 6.24.10 [Process management], page 318), for the detailed description of detached argument.

**host hostspec**  
[Subprocess argument]  
This argument is used to execute command on the remote host. The full syntax of hostspec is protocol:user@hostname:port, where protocol:, user@, or :port part can be omitted.

The protocol part specifies the protocol to communicate with the remote host; currently only ssh is supported, and it is also the default when protocol is omitted. The user part specifies the login name of the remote host. The hostname specifies the remote host name, and the port part specifies the alternative port number which protocol connects to.

The command line arguments are interpreted on the remote host. On the other hand, the I/O redirection is done on the local end. For example, the following code reads the file /foo/bar on the remote machine and copies its content into the local file baz in the current working directory.

```scheme
(do-process '(cat "bar")
 :host "remote-host.example.com"
 :directory "/foo"
 :output "baz")
```
9.27.2 Running process pipeline

**do-pipeline** commands :key input output error directory sigmask  
(on-abnormal-exit)

**run-pipeline** commands :key input output error wait directory sigmask

{gauche.process} Convenience routines to run pipeline of processes at once. Example:

```
(do-pipeline '((ls "src/")
  (grep "\..c$")
  (wc -l)))
```

This is equivalent to shell command pipeline `ls src/ | grep \.c$ | wc -l`, i.e. shows the number of C source files in the `src` subdirectory.

The `commands` argument is a list of lists. Each list must be `cmd/args` argument `do-process/run-process` can accept. At least one command must be specified.

The specified commands will run concurrently, with the stdout of the first command is connected to the stdin of the second, and stdout of the second to the stdin of the third, and so on. The stdin of the first command is fed from the source specified by the `input` keyword argument, and the stdout of the last command is sent to the sink specified by the `output` keyword argument. The default values of these are the calling process’s stdin and stdout, respectively. See `do-process/run-process`, for the possible values of these arguments (see Section 9.27.1 [Running subprocess], page 488).

The stderr of all the processes are sent to the sink specified by the `error` keyword argument, which is defaulted by the calling process’s stderr.

Like `do-process`, `do-pipeline` waits for completion of all the processes, and returns `#t` if the tail process succeeds (i.e. exits with zero status) or `#f` if the last process fails (i.e. exits with non-zero status). If you give `:error` to `on-abnormal-exit` keyword arguments, however, a failure of the tail process raises an error. Exit statuses of subprocesses other than the tail one are collected by `process-wait`, but won’t affect the return value, and won’t cause an error even `on-abnormal-exit` is `:error`.

On the other hand, `run-pipeline` returns a `<process>` object of the tail process. You can get other process objects in the pipeline by applying `process-upstreams` to the tail process. By default, `run-pipeline` runs all the subprocesses in background and returns immediately. Calling `process-wait` on the returned process object will waits for all the subprocesses. If you give a true value to `wait` keyword argument, `run-process` waits for all the subprocesses to finish before returning.

The `directory` and `sigmask` keyword arguments are applied to all the processes; see `do-process/run-process` for the description of these arguments (see Section 9.27.1 [Running subprocess], page 488).

Note: In Gauche 0.9.5, we introduced `run-process-pipeline`. It is similar to the current `run-pipeline` but returns a list of subprocess objects instead of a single one. We realized it’s not very convenient, so we deprecated `run-process-pipeline` and replaced it with `run-pipeline`. We still support `run-process-pipeline`, but strongly recommend to move to `run-pipeline` as soon as possible.

9.27.3 Process object

**<process>**  
{gauche.process} An object to keep the status of a child process. You can create the process object by `run-process` procedure described below. The process ports explained in the next section also use process objects.
The `<process>` class keeps track of the child processes spawned by high-level APIs such as `run-process` or `open-input-process-port`. The exit status of such children must be collected by `process-wait` or `process-wait-any` calls, which also do some bookkeeping. Using the low-level process calls such as `sys-wait` or `sys-waitpid` directly will cause inconsistent state.

**<process-abnormal-exit>**  
{gauche.process} A condition type mainly used by the process port utility procedures. Inherits `<error>`. This type of condition is thrown when the high-level process port utilities detect the child process exited with non-zero status code.

**process**  
[Instance Variable of `<process-abnormal-exit>`]

A process object.

Note: In Unix terms, exiting a process by calling `exit(2)` or returning from `main()` is a normal exit, regardless of the exit status. Some commands do use non-zero exit status to tell one of the normal results of execution (such as `grep(1)`). However, large number of commands uses non-zero exit status to indicate that they couldn’t carry out the required operation, so we treat them as exceptional situations.

**process? obj**  
[Function]

{gauche.process} ≡ (is-a? obj <process>)

**process-pid (process <process>)**  
[Method]

{gauche.process} Returns the process ID of the subprocess `process`.

**process-command (process <process>)**  
[Method]

{gauche.process} Returns the command invoked in the subprocess `process`.

**process-input (process <process>) :optional name**  
[Method]

**process-output (process <process>) :optional name**  
[Method]

{gauche.process} Retrieves one end of a pipe, whose another end is connected to the process’s input or output, respectively. `name` is a symbol given to the `redirects` argument of `run-process` to distinguish the pipe. See the following example:

```scheme
(let1 p (run-process ’(command arg)
   :redirects ‘((< 3 aux-in)
             (> 4 aux-out)))
  (let ([auxin (process-input p ’aux-in)]
       [auxout (process-output p ’aux-out)])
    ;; feed something to the child’s input
    (display ’something auxin)
    ;; read data from the child’s output
    (read-line auxout)
    ...
  )
  (process-wait p))
```

The symbols `aux-in` and `aux-out` is used to identify the pipes. Note that `process-input` returns `output` port, and `process-output` returns `input` port.

When `name` is omitted, `stdin` is used for `process-input` and `stdout` is used for `process-output`. These are the names used if child’s stdin and stdout are redirected by `:input :pipe` and `:output :pipe` arguments, respectively.

If there’s no pipe with the given name, `#f` is returned.

```scheme
(let* ((process (run-process ’("date") :output :pipe)))
```
If `process` is a result of `run-pipeline`, `(process-input process)` and `(process-input process 'stdin)` behave slightly differently—they return the pipe connected to the stdin of the head process of the pipeline, not the process represented by `process` (which is the tail of the pipeline). This allows you to treat the whole pipeline as one entity.

```
(let1 p (run-pipeline '((cat)
  (grep "aba")
  :input :pipe :output :pipe)
  (display "banana\nhabana\ntabata\ncabara\n"
  (process-input p)) ; head of the pipeline
  (close-port (process-input p))
  (process-wait p)
  (port->string (process-output p)))
⇒ "habana\ntabata\ncabara\n"
```

**process-error** (process <process>)  
{gauche.process} This is equivalent to `(process-output process 'stderr)`.

**process-alive?** process  
{gauche.process} Returns true if `process` is alive. Note that Gauche can’t know the subprocess’ status until it is explicitly checked by `process-wait`.

**process-upstreams** process  
{gauche.process} If `process` is the result of `run-pipeline`, this returns a list of processes that are upstream of `process` in the pipeline. If `process` is not the result of `run-pipeline`, this returns an empty list.

```
(define p (run-pipeline '((cat) (grep "ho") (wc)) :input :pipe))

p ⇒ #<process 20658 "wc" active>

(process-upstreams p)
⇒ (#<process 20656 "cat" active> #<process 20657 "grep" active>)
```

**process-list**  
{gauche.process} Returns a list of active processes. The process remains active until its exit status is explicitly collected by `process-wait`. Once the process’s exit status is collected and its state changed to inactive, it is removed from the list `process-list` returns.

**process-wait** process :optional nohang error-on-nonzero-status  
{gauche.process} Obtains the exit status of the subprocess `process`, and stores it to `process`’s status slot. The status can be obtained by `process-exit-status`.

This suspends execution until `process` exits by default. However, if a true value is given to the optional argument `nohang`, it returns immediately if `process` hasn’t exit.

If a true value is given to the optional argument `error-on-nonzero-status`, and the obtained status code is not zero, this procedure raises `<process-abnormal-exit>` error.

Returns `#t` if this call actually obtains the exit status, or `#f` otherwise.

If the process object is created by `run-pipeline` (see Section 9.27.2 [Running process pipeline], page 494), `process-wait` waits all of the subprocesses in the pipeline, not just
the last one, unless true value is given to the nohang argument. However, error-on-nonzero-status only affects to the status of process, which represents the last process in the pipeline; if an other subprocess exits with nonzero status, it is stored in its respective process objects, but won’t cause a fuss.

If you specify a true value to nohang for the pipelined process, process-wait still probes other subprocesses in the pipeline and updates exit statuses of terminated ones, but doesn’t wait unterminated subprocesses. The unterminated subprocesses should be waited individually, or by process-wait-any, to collect their exit statuses.

**process-wait-any :optional nohang**

*{gauche.process}* Obtains the exit status of any of the subprocesses created by run-process. Returns a process object whose exit status is collected.

If a true value is given to the optional argument nohang, this procedure returns #f immediately even if no child process has exit. If nohang is omitted or #f, this procedure waits for any of children exits.

If there’s no child processes, this procedure immediately returns #f.

**process-wait/poll**

*process :key interval max-wait continue-test raise-error?*

*{gauche.process}* Polls wait status of process periodically, sleeping interval nanoseconds (default 2e6 ns, i.e. 2ms) between each poll. It returns as soon as it finds the process has exited and its status is retrieved.

If max-wait is given and not #f, it species maximum duration in nanoseconds to keep polling. Once the duration expires, the procedure gives up and return. If it is not given or #f, the procedure keeps polling until the process exits.

The continue-test, if given and not #f, must be a procedure that takes one argument, poll count. It is called after each unsuccessul polling (that is, the process hasn’t been exitted), and the argument begins with 0 and incretemented for each callback. The procedure can return #f to give up polling. When it returns a true value, polling continues.

The raise-error? argument is the same as process-wait; an error is raised if process’s exit status isn’t zero.

The procedure returns #t if the process exitrd, and #f if it has given up. The process’s exit status should be retrieved with process-exit-status from process.

**process-exit-status**

*process*

*{gauche.process}* Returns exit status of process retrieved by process-wait. If this is called before process-wait is called on process, the result is undefined.

The meaning of exit status depends on the platform. You need to use sys-wait-exited? or sys-wait-signaled? to see if it is terminated voluntarily or by a signal, and use sys-wait-exit-status or sys-wait-termsig to extract the exit code or the terminating signal (see Section 6.24.10 [Process management], page 318).

**process-send-signal**

*process signal*

*{gauche.process}* Sends a signal signal to the subprocess process. signal must be an exact integer for signal number. See Section 6.24.7 [Signal], page 306, for predefined variables of signals.

**process-kill**

*process*

**process-stop**

*process*

**process-continue**

*process*

*{gauche.process}* Sends SIGKILL, SIGSTOP and SIGCONT to process, respectively.
process-shutdown  process :key ask ask-interval ask-retry signals  

Tries to terminate process in the gradually escalating means.

The ask argument is, if not #f, a procedure taking one argument, the retry count. The ask procedure is responsible to interact with process in a proper protocol (e.g. send exit command via communication channel). After ask is executed (the return value is ignored), the process’s status is monitored, and process-shutdown returns #t as soon as the process exits. If the process doesn’t exit, the procedure repeats calling ask up to retry times, with ask-interval nanoseconds delay inbetween, incrementing the retry count argument. If ask argument is #f, this step is skipped. The default value of ask is #f, ask-interval is 50e6 ns (50ms), and ask-retry is 1.

If the first step fails, this procedure starts sending signals. The process’s status is checked every time after a signal is sent. As soon as the process exists, process-shutdown returns #t. The sequence of signals is specified by signals argument, and the interval of sending signals is specified by signal-interval argument. The default value of signals is (list SIGTERM SIGTERM SIGKILL), and singal-interval is 50e6 ns (50ms).

If all the measures fail, #f is returned.

(Hint: If you wants to keep sending signals until the process really exits, you can pass a circular list to the signals.)

9.27.4 Process ports

open-input-process-port  command :key input error encoding  

{gauche.process} Runs command asynchronously in a subprocess. Returns two values, an input port which is connected to the stdout of the running subprocess, and a process object. Command can be a string, a list of command name and arguments, or a list of lists of command name and arguments.

If it is a string, it is passed to /bin/sh. You can use shell metacharacters in this form, such as environment variable interpolation, globbing, and redirections. If you create the command line by concatenating strings, it’s your responsibility to ensure escaping special characters if you don’t want the shell to interpret them. The shell-escape-string function described below might be a help.

If command is a list (but not a list of lists), each element is converted to a string by x->string and then passed directly to sys-exec (the car of the list is used as both the command path and the first element of argv, i.e. argv[0]). Use this form if you want to avoid the shell from interfering; i.e. you don’t need to escape special characters.

The subprocess’s stdin is redirected from /dev/null, and its stderr shares the calling process’s stderr by default. You can change these by giving file pathnames to input and error keyword arguments, respectively.

If command is a list of lists, it creates a command pipeline, as in run-pipeline (see Section 9.27.2 [Running process pipeline], page 494). Each inner list should consists of a command path, followed by command-line arguments. They are applied on x->string before passed to sys-exec. The stdout of the last command is available from the returned input port, and the stdin of the first command is provided by input keyword arguments, or /dev/null by default. The stderr of all the commands goes to error keyword arguments if given, or shared with the caller process’s stderr.

You can also give the encoding keyword argument to specify character encoding of the process output. If it differs from the Gauche’s internal encoding format, open-input-process-port inserts a character encoding conversion port. If encoding is given, the conversion-buffer-size
keyword argument can control the conversion buffer size. See Section 9.5 [Character code conversion], page 399, for the details of character encoding conversions.

(receive (port process) (open-input-process-port "ls -l Makefile")
  (begin0 (read-line port)
    (process-wait process)))
⇒ "-rw-r--r-- 1 shiro users 1013 Jun 22 21:09 Makefile"

(receive (port process) (open-input-process-port '(ls -l "Makefile"))
  (begin0 (read-line port)
    (process-wait process)))
⇒ "-rw-r--r-- 1 shiro users 1013 Jun 22 21:09 Makefile"

(open-input-process-port "command 2>&1")
⇒ ;the port reads both stdout and stderr

(open-input-process-port "command 2>&1 1>/dev/null")
⇒ ;the port reads stderr

The exit status of subprocess is not automatically collected. It is the caller’s responsibility to issue process-wait, or the subprocess remains in a zombie state. If it bothers you, you can use one of the following functions.

call-with-input-process command proc :key input error encoding
  conversion-buffer-size on-abnormal-exit
{gauche.process} Runs command in a subprocess and pipes its stdout to an input port, then call proc with the port as an argument. When proc returns, it collects its exit status, then returns the result proc returned. The cleanup is done even if proc raises an error.

The keyword argument on-abnormal-exit specifies what happens when the child process exits with non-zero status code. It can be either :error (default), :ignore, or a procedure that takes one argument. If it is :error, a <process-abnormal-exit> error condition is thrown by non-zero exit status; the process slot of the condition holds the process object. If it is :ignore, nothing is done for non-zero exit status. If it is a procedure, it is called with a process object; when the procedure returns, call-with-input-process returns normally.

The semantics of command and other keyword arguments are the same as open-input-process-port above.

(call-with-input-process "ls -l *
  (lambda (p) (read-line p)))

with-input-from-process command thunk :key input error encoding
  conversion-buffer-size on-abnormal-exit
{gauche.process} Runs command in a subprocess, and calls thunk with its current input port connected to the command’s stdout. The command is terminated and its exit status is collected, after thunk returns or raises an error.

The semantics of command and keyword arguments are the same as call-with-input-process above.

(with-input-from-process "ls -l * read-line)

open-output-process-port command :key output error encoding
  conversion-buffer-size
{gauche.process} Runs command in a subprocess asynchronously. Returns two values, an output port which is connected to the stdin of the subprocess, and the process object.

The semantics of command is the same as open-input-process-port. The semantics of encoding and conversion-buffer-size are also the same.
The subprocess’s stdout is redirected to /dev/null by default, and its stderr shares the calling process’s stderr. You can change these by giving file pathnames to output and error keyword arguments, respectively.

The exit status of the subprocess is not automatically collected. The caller should call process-wait on the subprocess at appropriate time.

call-with-output-process command proc :key output error encoding  
{gauche.process} Runs command in a subprocess, and calls proc with an output port which is connected to the stdin of the command. The exit status of the command is collected after either proc returns or raises an error.

The semantics of keyword arguments are the same as open-output-process-port, except on-abnormal-exit, which is the same as described in call-with-input-process.

(call-with-output-process "/usr/sbin/sendmail" 
(lambda (out) (display mail-body out)))

with-output-to-process command thunk :key output error encoding  
{gauche.process} Same as call-with-output-process, except that the output port which is connected to the stdin of the command is set to the current output port while executing thunk.

call-with-process-io command proc :key error encoding  
{gauche.process} Runs command in a subprocess, and calls proc with two arguments; the first argument is an input port which is connected to the command’s stdout, and the second is an output port connected to the command’s stdin. The error output from the command is shared by the calling process’s, unless an alternative pathname is given to the error keyword argument.

The exit status of the command is collected when proc returns or raises an error.

process-output->string command :key error encoding  
{gauche.process} Runs command and collects its output (to stdout) and returns them. process-output->string concatenates all the output from command to one string, replacing any sequence of whitespace characters to single space. The action is similar to “command substitution” in shell scripts. process-output->string-list collects the output from command line-by-line and returns the list of them. Newline characters are stripped.

Internally, command is run by call-with-input-process, to which keyword arguments are passed.

(Tip: To receive stderr output of the child process in the result as well, pass :merge to :error. See run-process above for the details.)

(process-output->string '(uname -smp))
⇒ "Linux i686 unknown"

(process-output->string '(ls))
⇒ "a.out foo.c foo.c~ foo.o"

(process-output->string-list '(ls))
⇒ ("a.out" "foo.c" "foo.c~" "foo.o")
shell-escape-string str :optional flavor

{gauche.process} If str contains characters that affects shell’s command-line argument parsing, escape str to avoid shell’s interpretation. Otherwise, returns str itself.

The optional flavor argument takes a symbol to specify the platform; currently windows and posix can be specified. The way shell handles the escape and quotation differ a lot between these platforms; the windows flavor uses MSVC runtime argument parsing behavior, while the posix flavor assumes IEEE Std 1003.1. When omitted, the default value is chosen according to the running platform. (Note: Cygwin is regarded as posix.)

Use this procedure when you need to build a command-line string by yourself. (If you pass a command-line argument list, instead of a single command-line string, you don’t need to escape them since we bypass the shell.)

shell-tokenize-string str :optional flavor

{gauche.process} Split a string str into arguments as the shell does.

```
(shell-tokenize-string "grep -n -e "foo bar" log")
⇒ ("grep" "-n" "-e" "foo bar" "log")
```

The optional flavor arguments can be a symbol either windows or posix to specify the syntax. If it’s windows, we follow MSVC runtime command-line argument parser behavior. If it’s posix, we follow IEEE Std 1003.1 Shell Command Language. When omitted, the default value is chosen according to the running platform. (Note: Cygwin is regarded as posix.)

This procedure does not handle fancier shell features such as variable substitution. If it encounters a metacharacter that requires such interpretation, an error is signaled. In other words, metacharacters must be properly quoted in str.

```
(shell-tokenize-string "echo $foo" 'posix)
⇒ signals error
```

```
(shell-tokenize-string "echo "$foo"" 'posix)
⇒ still signals error
```

```
(shell-tokenize-string "echo "$foo"" 'posix)
⇒ ("echo" "$foo")
```

```
(shell-tokenize-string "echo \$foo" 'posix)
⇒ ("echo" "$foo")
```

9.27.5 Process connection

<process-connection> [Class]

A connection abstraction to communicate with an external process. Inherits <connection>. See Section 9.9 [Connection framework], page 426, for the details of the connection interface.

This is useful to give an external process to the code that expects connection. For example, instead of direct network connection, you can insert a filter process between remote server and your client code.

make-process-connection process-or-spec [Function]

Run an external process and returns a connection that’s connected to standard I/O of the process.

You can pass a list of command and its arguments, or a <process> object, to the process-or-spec argument. If you it’s a <process> object, it’s stdin and stdout must be connected to pipes. If it is a list, it is passed to run-process to run a new process.
Shutting down both channels of the connection terminates the process. Most processes that
reads from stdin would exits after it reads EOF from input, so we just poll the process exit
status for a short period of time. If the process doesn’t exit, we send signals (first SIGTERM,
then SIGKILL) to ensure the termination of the process.
Merely closing the connection doesn’t terminate the process, so that the forked process can
keep talking to the process.

9.28 gauche.record - Record types

gauche.record

This module provides a facility to define record types, user-defined aggregate types. The API
is upper compatible to SRFI-9 (Defining Record Types) and SRFI-99 (ERR5RS Records).

Record types are implemented as Gauche’s classes, but have different characteristics from
the general classes. See Section 9.28.1 [Record types introduction], page 502, for when you want
to use record types.

The syntactic layer is the define-record-type macro that conveniently defines a record
type and related procedures (a constructor, a predicate, accessors and modifiers) all at once
declaratively. Knowing this macro alone is sufficient for most common usage of records.

The inspection layer defines common procedures to query information to the records and
record types.

The procedural layer is a low-level machinery to implement the syntactic layer; you don’t
usually need to use them in day-to-day programming, but they might be handy to create record
types on-the-fly at runtime.

9.28.1 Introduction

Gauche provides a general way for users to define new types as new classes, using object system
(see Chapter 7 [Object system], page 329), and indeed record types are implemented as Gauche’s
classes. However, using record types instead of classes has several advantages.

• It is portable. The API conforms two major record SRFIs, SRFI-9 and SRFI-99, so the
code using record types can run on various Scheme systems.

• It is efficient. Record types are less flexible than classes, but that allows Gauche to optimize
more. Hence creating records and accessing/modifying them are much faster than creating
instances of general classes and accessing/modifying them. It makes record types preferable
choice when you only need a mechanism to bundle several related values to carry around,
and don’t need fancier mechanisms such as class redefinitions.

• As Gauche’s extension, you can define pseudo record types, which interprets ordinary ag-
gregate types such as vectors and lists as records. (For Common Lisp users; it is like the
:type option of defstruct). This helps flexibility of interface. For example, you can ask
your library’s users to pass a point in a vector of three numbers, instead of asking users to
pack their point data into your custom point record type. Yet inside your library you can
treat the passed data as if it is your point record type. See Section 9.28.5 [Pseudo record
types], page 507, for more details.

The disadvantage of record types is that they don’t obey Gauche’s class redefinition protocol
(see Section 7.2.5 [Class redefinition], page 342). That is, if you redefine a record with the same
name, it creates a new record type unrelated to the old one. The record instances created from
the old definition won’t be updated according to the new definition.

More importantly, record constructors, accessors and modifiers are tend to be inlined where
they are used, to achieve better performance. Since they are inlined, the code that uses those
procedures are not affected when the record type is redefined. This means if you redefine a
record type, you have to reload (recompile) the sources that uses any of record constructors,
accessors or modifiers.

9.28.2 Syntactic Layer

**define-record-type** type-spec ctor-spec pred-spec field-spec . . .  

[Macro]  
[R7RS base][SRFI-9][SRFI-99*] {gauche.record} Defines a record type, and optionally de-

fines a constructor, a predicate, and field accessors and modifiers.
The *type-spec* argument names the record type, and optionally specifies the supertype (*parent*).

```
(type-spec : type-name
           | (type-name parent option ...))
```

The *type-name* identifier will be bound to a record type descriptor, or rtd, which can be used
for introspection and reflection. See Section 9.28.3 [Record types inspection layer], page 505,
and Section 9.28.4 [Record types procedural layer], page 506, for possible operations for record
types. In Gauche, a record type descriptor is a `<class>` with a metaclass `<record-meta>`.
The *parent* expression should evaluate to a record type descriptor. If given, the defined record
type inherits it; that is, all the slots defined in the parent type are available to the *type-name*
as well, and the instance of *type-name* answers `#t` to the predicate of the parent type.

Since a record type is also a class, parent type is also a superclass of the defined record type.
However, record types are limited to have single implementation inheritance. The parent type
must be a subclass of `<record>`. (You can have abstract classes along the main inheritance,
though. See mixins below.)

You can give a pseudo record base type as *parent* to define a pseudo record type, which allows
you to access ordinary aggregates like vectors as records. See Section 9.28.5 [Pseudo record
types], page 507, for more details.

the *option* . . . part is Gauche’s extension. It must be a keyword-value list. The following
keywords are recognized:

`:mixins (class ...)
  Specifies auxiliary superclasses. The classes must be abstract, that is, must not
  have slots. It is to implement protocols in the record type, e.g. `<sequence>` (see
  Section 9.31 [Sequence framework], page 510).

`:metaclass metaclass
  Specifies alternative metaclass. By default, metaclass of record types is
  `<record-meta>`. If you specify an alternative metaclass, it must be a subclass
  of `<record-meta>`.

The *ctor-spec* defines the constructor of the record instance.

```
(ctor-spec : #f | #t | ctor-name
           | (ctor-name field-name ...))
```

If it is `#f`, no constructor is created. If it is `#t`, a default constructor is created with a name
`make-type-name`. If it is a single identifier *ctor-name*, a default constructor is created with
the name. The default constructor takes as many arguments as the number of fields of the
record, including inherited ones if any. When called, it allocates an instance of the record,
and initialize its fields with the given arguments in the order (inherited fields comes first),
and returns the record.

The last variation of `ctor-spec` creates a custom constructor with the name `ctor-name`. The
custom constructor takes as many arguments as the given `field-names`, and initializes
the named fields. If the inherited record type has a field of the same name as the ancestor record
type, only the inherited ones are initialized. In Gauche, uninitialized fields remains unbound
until some value is set to it.

The `pred-spec` defines the predicate of the record instance, which takes one argument and
returns `#t` if it is an instance of the defined record type or its descendants.

```
pred-spec : #f | #t | pred-name
```

`pred-name` : identifier

If it is `#f`, no predicate is created. If it is `#t`, a predicate is created with a name `type-name?`.
If it is a single identifier, a predicate is created with the given name.

The rest of the arguments specify fields (slots) of the record.

```
field-spec
 : field-name ; immutable, with default accessor
 | (field-name) ; mutable, with default accessor/modifier
 | (field-name accessor-name); immutable
 | (field-name accessor-name modifier-name); mutable
```

`field-name` : identifier
`accessor-name` : identifier
`modifier-name` : identifier

The first and the third forms define immutable fields, which can only be initialized by the
constructor but cannot be modified afterwards (thus such fields don’t have modifiers). The
second and the fourth forms define mutable fields.

The third and fourth forms explicitly name the accessor and modifier. With the first and
second forms, on the other hand, the accessor is named as `type-name-field-name`, and the
modifier is named as `type-name-field-name-set!`.

Let’s see some examples. Here’s a definition of a record type `point`.

```
(define-record-type point #t #t
  x y z)
```

The variable `point` is bound to a record type descriptor, which is just a class. But you can
take its class and see it is indeed an instance of `<record-meta> metaclass.

```
point ⇒ #<class point>
(class-of point) ⇒ #<class <record-meta>>
```

You can create an instance of `point` by the default constructor `make-point`. The predicate
is given the default name `point?`, and you can access the fields of the created record by `point-x
etc.

```
(define p (make-point 1 2 3))
```

```
(point? p) ⇒ #t
(point-x p) ⇒ 1
(point-y p) ⇒ 2
(point-z p) ⇒ 3
```
Since we defined all fields immutable, we cannot modify the instance \texttt{p}.

Here's a mutable version of point, \texttt{mpoint}. You can modify its fields by modifier procedures and generalized \texttt{set!}.

\begin{verbatim}
(define-record-type mpoint #t #t
 (x) (y) (z))

(define p2 (make-mpoint 1 2 3)); create an instance

(mpoint-x p2) ⇒ 1

(mpoint-x-set! p2 4); default modifier
(mpoint-x p2) ⇒ 4

(set! (mpoint-x p2) 6); generalized \texttt{set!} also works
(mpoint-x p2) ⇒ 6
\end{verbatim}

Next one is an example of inheritance. Note that the default constructor takes arguments for fields of the parent record as well.

\begin{verbatim}
(define-record-type (qpoint mpoint) #t #t
 (w))

(define p3 (make-qpoint 1 2 3 4))

(qpoint? p3) ⇒ #t ; p3 is a qpoint
(mpoin? p3) ⇒ #t ; ... and also an mpoint

(mpoint-x p3) ⇒ 1 ; accessing inherited field
(mpoint-y p3) ⇒ 2
(mpoint-z p3) ⇒ 3
(qpoint-w p3) ⇒ 4
\end{verbatim}

A small caveat: Accessors and modifiers for inherited fields (e.g. \texttt{qpoint-x} etc.) are not created.

Gauche's convention is to enclose class name by \texttt{<>}. You can follow the convention and still explicitly gives simpler names (instead of \texttt{make-<point>} or \texttt{<point>-x}):

\begin{verbatim}
(define-record-type <point> make-point point?
 (x point-x)
 (y point-y)
 (z point-z))
\end{verbatim}

### 9.28.3 Inspection layer

This layer provides common procedures that operates on record type descriptors and record instances.

Note that a record type descriptor is a class in Gauche, so you can also use operators on classes (e.g. \texttt{class-name}, \texttt{class-slots} etc.) on record type descriptors as well. However, these procedures are more portable.

\begin{verbatim}
record? obj
[SRFI-99][R6RS]{gauche.record} Returns \#t iff \texttt{obj} is an instance of record type, \#f otherwise.

record-rtd record
[SRFI-99][R6RS]{gauche.record} Returns the record type descriptor of the record instance.
\end{verbatim}
Chapter 9: Library modules - Gauche extensions

**rtd-name rtd**

- [Function]
  - [SRFI-99]{gauche.record} Returns the name of the record type descriptor rtd.

**rtd-parent rtd**

- [Function]
  - [SRFI-99]{gauche.record} Returns the parent type of the record type descriptor rtd. If rtd doesn’t have a parent, #f is returned.

**rtd-field-names rtd**

- [Function]
  - [SRFI-99]{gauche.record} Returns a vector of symbols, each of which is the names of the direct fields of the record represented by rtd. The result doesn’t include inherited fields.

**rtd-all-field-names rtd**

- [Function]
  - [SRFI-99]{gauche.record} Returns a vector of symbols, each of which is the names of the fields of the record represented by rtd. The result includes all inherited fields.

**rtd-field-mutable? rtd field-name**

- [Function]
  - [SRFI-99]{gauche.record} Returns #t iff the field with the name field-name of a record represented by rtd is mutable.

### 9.28.4 Procedural layer

These procedures are low-level machinery on top of which define-record-type is implemented. They can be used to create a new record type at runtime.

**make-rtd name field-specs :optional parent**

- [Function]
  - [SRFI-99]{gauche.record} Creates and returns a new record type descriptor with name name and having fields specified by field-specs. If parent is given, it must be a record type descriptor or #f. If it is a record type descriptor, the created record type inherits from it.

  The field-specs argument must be a vector, each element of which is a field specifier. A field specifier can be a symbol, a list (mutable symbol), or a list (immutable symbol). The symbol names the field. A single symbol or (mutable symbol) format makes the field mutable, and (immutable symbol) format makes the field immutable.

  Note: Gauche does not implement the extension suggested in SRFI-99 yet, which is sealed, opaque and uid arguments.

**rtd? obj**

- [Function]
  - [SRFI-99]{gauche.record} Returns #t if obj is a record type descriptor, #f otherwise.

**rtd-constructor rtd :optional field-specs**

- [Function]
  - [SRFI-99]{gauche.record} Returns a procedure that creates an instance record of the record type represented by rtd. Without field-specs, it returns the default constructor, which takes as many arguments as the number of fields of the record to initialize them.

  You can give a vector of symbols as field-specs. The n-th symbol specifies which field of the instance should be initialized by the n-th argument. The field-specs vector cannot contain duplicate names. If the record type defines a field with the same name as the one in the parent record type, the custom constructor can only initialize the field of the derived type’s instance.

**rtd-predicate rtd**

- [Function]
  - [SRFI-99]{gauche.record} Returns a predicate to test an object is an instance of rtd.

  If rtd is a pseudo record type, the predicate merely tests the given object is in an appropriate type and has enough size to hold the contents. See Section 9.28.5 [Pseudo record types], page 507, for the details.
rtd-accessor  rtd field-name  [Function]
[SRFI-99]{gauche.record} Returns a procedure that takes one argument, an instance of rtd, and returns the value of the field-name of the instance.

An error is signaled if the record type doesn’t have the field of name field-name.

If rtd is inherits other record types, and it defines a field of the same name as inherited ones, then the accessor returned by this procedure retrieves the value of the field of the derived record.

rtd-mutator  rtd field-name  [Function]
[SRFI-99]{gauche.record} Returns a procedure that takes two arguments, an instance of rtd and a value, and sets the latter as the value of the field-name of the instance.

An error is signaled if the record type doesn’t have the field of name field-name, or the named field is immutable.

Like rtd-accessor, if the record has a field with the same name as inherited one, the modifier returned by this procedure only modifies the field of the derived record.

9.28.5 Pseudo record types

A pseudo record type is a record type that does not create an instance of its own type. Instead it treats an object of other collection types, such as a vector, as if it had named fields. It’s easier to understand by an example:

```scheme
(define-record-type (vpoint (pseudo-rtd <vector>)) #t #t
  (x) (y) (z))
(make-vpoint 1 2 3) ⇒ #(1 2 3)
(vpoint-x '#(1 2 3)) ⇒ 1
(rlet1 v (make-vpoint 1 2 3)
  (set! (vpoint-y v) -1))
⇒ #(1 -1 3)
```

To create a pseudo record type, specify another pseudo record type as a parent. The procedure pseudo-rtd can be used to obtain a base pseudo record type of the suitable instance class.

pseudo-rtd  instance-class  [Function]
{gauche.record} Returns a pseudo rtd suitable to use instance-class as a pseudo record.

Currently, <list>, <vector>, and uniform vector classes (<u8vector> etc.) are supported as instance-class.

The predicates of a pseudo record return #t if the given object can be interpreted as the pseudo record. In the above example of vpoint record, the predicate vpoint? returns #t iff the given object is a vector with 3 or more elements:

```scheme
(vpoint? '#(0 0 0)) ⇒ #t
(vpoint? '#(0 0)) ⇒ #f
(vpoint? '(0 0 0)) ⇒ #f
(vpoint? '#(0 0 0 0)) ⇒ #t
```

We allow more elements so that the pseudo record can be used to interpret the header part of the longer data.
9.29 \texttt{gauche.reload} - Reloading modules

\texttt{gauche.reload} \hspace{1cm} [Module]

In the development cycle, you often have to reload modules frequently. This module supports it.

Note that some part of semantics of the program depends on the order of loading modules, so reloading arbitrary modules may change the program behavior unexpectedly. This module is for developers who knows what they are doing.

Redefinition rules: Reloading a module resets all the binding in the module by default. Sometimes it is not desirable, however. For example, you might want to keep an intermediate results in some variable. You can specify rules for the reloading procedure to determine which binding to keep.

The rule is described in the following syntax.

\begin{verbatim}
<module-rules> : (<module-rule> ...)
<module-rule> : (<module-pattern> <rule> ...)
<module-pattern> : a symbol module name, or a symbol containing glob pattern
<rule> : procedure | symbol | regexp
    | (and <rule> ...)
    | (or <rule> ...)
    | (not <rule>)
\end{verbatim}

\texttt{<module-rules>} is the global rule to determine per-module rules. \texttt{<module-pattern>} is either a symbol module name or a symbol that contains glob pattern (e.g. \texttt{mylib.*}). If \texttt{<rule>} is a procedure, it is used as a predicate and the bindings whose value satisfies the predicate are kept from redefinition. If \texttt{<rule>} is a symbol, the binding of the variable whose name is the symbol is kept. If \texttt{<rule>} is a regexp, the bindings of the variable whose name matches the regexp are kept.

Note that the mechanism to prevent redefinition is kind of ad-hoc hack and semantically unclean. Especially, the right-hand expressions of \texttt{defines} are still evaluated, so any side effects they have will be in effect (e.g. \texttt{define-class} would still redefine a class). It’s just for your convenience. Take a look at the code if you want to know the exact behavior.

\texttt{reload module-name :optional rule ...} \hspace{1cm} [Function]

\{\texttt{gauche.reload}\} Reloads the specified module. You can optionally specify redefinition rules by \texttt{rule} \ldots, where each \texttt{rule} is the term \texttt{<rule>} defined above.

\texttt{reload-modified-modules :optional module-rules} \hspace{1cm} [Function]

\{\texttt{gauche.reload}\} Reloads module(s) that have been modified since they are loaded last time. If optional \texttt{module-rules} is given, it is used to determine the redefinition rules for reloaded modules. If \texttt{module-rules} is omitted, the current rules are used. The default of current rules is empty. You can set the current rules by \texttt{module-reload-rules}.

\texttt{module-reload-rules :optional module-rules} \hspace{1cm} [Function]

\{\texttt{gauche.reload}\} This is a parameter (see Section 6.16.1 [Parameters], page 233) that keeps the default module rules for \texttt{reload-modified-modules}. If called without arguments, returns the current module rules. If called with \texttt{module-rules}, sets the argument to the current module rules.

\texttt{reload-verbose :optional flag} \hspace{1cm} [Function]

\{\texttt{gauche.reload}\} This is a parameter to control verbosity of the reloading procedures. If called without arguments, returns the current verbosity flag. If called with \texttt{flag}, it is set to the current verbosity flag.
9.30 gauche.selector - Simple dispatcher

This module provides a simple interface to dispatch I/O events to registered handlers, based on sys-select (see Section 6.24.11 [I/O multiplexing], page 321).

A dispatcher instance that keeps watching I/O ports with associated handlers. A new instance can be created by make method.

Add a handler proc to the selector. proc is called when port-or-fd, which should be a port object or an integer that specifies a system file descriptor, meets a certain condition specified by flags. flags must be a list of one or more of the following symbols.

r  Calls proc when data is available at port-or-fd to read.

w  Calls proc when port-or-fd is ready to be written.

x  Calls proc when an exceptional condition occurs on port-or-fd.

proc is called with two arguments. The first one is port-or-fd itself, and the second one is a symbol r, w or x, indicating the condition.

If a handler is already associated with port-or-fd under the same condition, the previous handler is replaced by proc.

Deletes the handler entries that matches port-or-fd, proc and flags. One or more of the arguments may be #f, meaning “don’t care”. For example,

(deletes all the handlers associated to the-port, and

(deletes all the handlers waiting for writable condition.

Dispatcher body. Waits for the conditions registered in self, and when it occurs, calls the associated handler. If the timeout argument is omitted or false, this method waits indefinitely. Alternatively you can give a timeout value, that can be a real number in microseconds, or a list of two integers that represents seconds and microseconds.

Returns the number of handlers called. Zero means the selector has been timed out.

It is safe to modify self inside handler. The change will be effective from the next call of selector-select

This is a simple example of "echo" server:

(use gauche.net)
(use gauche.selector)
(use gauche.uvector)

(define (echo-server port)
  (let ((selector (make <selector>))
        (server (make-server-socket 'inet port :reuse-addr? #t)))
    (define (accept-handler sock flag)
      (let* (((client (socket-accept server))

(output (socket-output-port client)))
(selector-add! selector
  (socket-input-port client :buffering #f)
  (lambda (input flag)
    (echo client input output))
  '(r))))

(define (echo client input output)
  (let ((str (read-uvector <u8vector> 4096 input)))
    (if (eof-object? str)
        (begin (selector-delete! selector input #f #f)
               (socket-close client))
        (begin (write-uvector str output)
               (flush output))))))

(selector-add! selector
  (socket-fd server)
  accept-handler
  '(r))
(do () (#f) (selector-select selector))))

9.31 gauche.sequence - Sequence framework

gauche.sequence [Module]
Provides a generic operations on sequences. A sequence is a collection with ordered elements. Besides all the operations applicable on collections, you can associate integer index to each element, and apply order-aware operations on the elements.

This module inherits gauche.collection (see Section 9.6 [Collection framework], page 404). All the collection generic operations can be applied to a sequence as well.

Among Gauche built-in class, lists, vectors and strings are sequences and the specialized methods are defined for them. Other extension types, such as uniform vectors, have the methods as well.

9.31.1 Fundamental sequence accessors

ref (seq <sequence>) index :optional fallback [Method]
{gauche.sequence} Returns index-th element of the sequence seq. This method enables uniform access for any sequence types.

When index is less than zero, or greater than or equal to the size of the sequence, fallback is returned if provided, or an error is signaled if not.

(ref '(a b c) 1) ⇒ b
(ref '#(a b c) 1) ⇒ b
(ref "abc" 1) ⇒ #\b

(setter ref) (seq <sequence>) index value [Method]
{gauche.sequence} Sets value to the index-th element of the sequence seq. This is the uniform sequence modifier.

Note: Some sequences may not support arbitrary modification by index. For example, if you have a sequence representing a set of sorted integers, you cannot modify i-th element with arbitrary value. Yet such sequence may provide other means of modification, such as inserting or deleting elements.

(let ((x (list 'a 'b 'c)))
```
(set! (ref x 1) 'z)
x) ⇒ (a z c)

(let ((x (vector 'a 'b 'c)))
  (set! (ref x 1) 'z)
  x) ⇒ #(a z c)

(let ((x (string #\a #\b #\c)))
  (set! (ref x 1) #\z)
  x) ⇒ "azc"
```

**Method**

**referencer** *(seq <sequence>)*

{gauche.sequence}

**Modifier** *(seq <sequence>)*

{gauche.sequence}

### 9.31.2 Slicing sequence

**subseq** *(seq <sequence>* :optional start end)*

{gauche.sequence} Retrieve a subsequence of the sequence seq, from start-th element (inclusive) to end-th element (exclusive). If end is omitted, up to the end of sequence is taken. The type of the returned sequence is the same as seq.

```
(subseq '(a b c d e) 1 4) ⇒ (b c d)
(subseq "#(a b c d e) 1 4) ⇒ #(b c d)
(subseq "abcde" 1 4) ⇒ "bcd"
(subseq '(a b c d e) 3) ⇒ (d e)
```

**setter subseq** *(seq <sequence>)*

{gauche.sequence} Sets the elements of value-seq from the start-th element (inclusive) to the end-th element (exclusive) of the sequence seq. Value-seq can be any sequence, but its size must be larger than (end - start).

In the second form, end is figured out by the length of value-seq.

```
(define s (vector 'a 'b 'c 'd 'e))
(set! (subseq s 1 4) '(4 5 6))
s ⇒ #(a 4 5 6 e)
(set! (subseq s 0) "ab")
s ⇒ #(\a \b 5 6 e)
```

### 9.31.3 Mapping over sequences

You can use extended fold, map, for-each and other generic functions on sequences, since a sequence is also a collection. However, sometimes you want to have index as well as the element itself during iteration. There are several generic functions for it.

```
fold-with-index kons knil (seq <sequence>)* ...
{gauche.sequence} Like generic fold, except kons is given the index within seq, as the first argument, as well as each element from seqs and the accrued value.

(fold-with-index acons '() '(a b c))
⇒ ((2 . c) (1 . b) (0 . a))
```
map-with-index proc (seq <sequence>) ...

Method

map-to-with-index class proc (seq <sequence>) ...

Method

for-each-with-index proc (seq <sequence>) ...

Method

{gauche.sequence} Like map, map-to and for-each, except proc receives the index as the first argument.

(map-with-index list '(a b c d) '(e f g h))
⇒ ((0 a e) (1 b f) (2 c g) (3 d h))

(map-to-with-index <vector> cons '(a b c d))
⇒ #((0 . a) (1 . b) (2 . c) (3 . d))

find-with-index pred (seq <sequence>)

Method

{gauche.sequence} Finds the first element in seq that satisfies pred like find, but returns two values, the index of the element and the element itself. If no element satisfies pred, two #f's are returned.

(find-with-index char-upper-case? "abraCadabra")
⇒ 4 and #\C

(find-with-index char-numeric? "abraCadabra")
⇒ #f and #f

find-index pred (seq <sequence>)

Method

{gauche.sequence} Like find, but returns the index of the first element that satisfies pred in seq, instead of the element itself. If no element in seq satisfies pred, #f is returned.

(find-index char-upper-case? "abraCadabra")
⇒ 4

(find-index char-numeric? "abraCadabra")
⇒ #f

See also list-index in scheme.list (see Section 10.3.1 [R7RS lists], page 593).

fold-right kons knil (seq <sequence>) ...

Method

{gauche.sequence} Generalization of fold-right on lists. Like fold, this method applies a higher-order function kons over given sequence(s), passing the "seed" value whose default is knil. The difference between fold and fold-right is the associative order of elements on which kons is applied.

When we have one sequence, [E0, E1, ..., En], fold and fold-right work as follows, respectively.

fold:
(kons En (kons En-1 (kons ... (kons E1 (kons E1 knil)) ...)))

fold-right
(kons E0 (kons E1 (kons ... (kons En-1 (kons En knil)) ...)))

This method isn’t defined on <collection>, since collections don’t care the order of elements.

9.31.4 Other operations over sequences

Selection and searching

sequence-contains haystack needle :key test

Generic function

{gauche.sequence} Both needle and haystack must be sequences. Searches needle from haystack from the beginning of haystack. If needle is found, the index in haystack where it
begins is returned. Otherwise \#f is returned. The keyword argument test is used to compare elements; its default is eqv?.

\[
\text{(sequence-contains '}(a b r a c a d a b r a) '}(b r a)) \\
\Rightarrow 1
\]

\[
\text{(sequence-contains '}(a b r a c a d a b r a) '}(c r a)) \\
\Rightarrow \#f
\]

This can be regarded as generalization of string-contains in SRFI-13 (see Section 11.6.7 [SRFI-13 String searching], page 697).

**break-list-by-sequence** list needle :key test

**break-list-by-sequence!** list needle :key test

{gauche.sequence} Searches a sequence needle from list, and if found, breaks list to two parts—the prefix of list up to right before needle begins, and the rest—and returns them. List must be a list, but needle can be any sequence. Elements are compared by test, defaulting to eqv?.

\[
\text{(break-list-by-sequence '}(a b r a c a d a b r a) '}(c a d)) \\
\Rightarrow (a b r a) \text{ and } (c a d a b r a)
\]

If needle isn’t found in list, it returns list itself and (). This behavior is aligned to span and break (see Section 10.3.1 [R7RS lists], page 593), which split a list by predicate but returns the whole list if split condition isn’t met.

\[
\text{(break-list-by-sequence '}(a b r a c a d a b r c a) '}(c a z)) \\
\Rightarrow (a b r a c a d a b r c a) \text{ and } ()
\]

The linear update version **break-list-by-sequence!** modifies list to create the return value if necessary, so list must be mutable. The caller must use the return value instead of relying on side-effects, though, for list may not be modified.

**sequence->kmp-stepper** needle :key test

{gauche.sequence} This is an internal routine to search subsequence (needle) inside larger sequence, using Knuth-Morris-Pratt (KMP) algorithm. It is used in sequence-contains, break-list-by-sequence and break-list-by-sequence!.

Returns a procedure that performs one step of KMP search. The procedure takes two arguments, an element elt and an index k. It compares elt with (‘ needle k), and returns two values—the next index and a flag indicating the match is completed. When the match is completed, the next index is equal to the length of needle.

As an edge case, if needle is an empty sequence, sequence->kmp-stepper returns \#f.

Elements are compared using test, which is defaulted to eqv?.

The following is a skeleton of searcher using sequence->kmp-stepper. Here we assume haystack is a list, and we just return whether the needle is found or not, or needle is empty; you might want to carry around other info in the loop (e.g. sequence-contains tracks the current index of haystack in order to return the found index.)

\[
\text{(if-let1 stepper (sequence->kmp-stepper needle)} \\
\text{(let loop ([haystack haystack] [k 0])} \\
\text{(if (null? haystack)} \\
\text{'not-found} \\
\text{(receive (k found) (stepper (car haystack) k)) ; KMP step} \\
\text{(if found} \\
\text{'found} \\
\text{(loop (cdr haystack) k))})} \\
\text{'needle-is-empty)}
\]
Note that selection and searching methods for collections can also be applied to sequences. See Section 9.6.2 [Selection and searching in collection], page 408.

**Grouping**

\[
\text{group-sequence \ seq \ :key \ key \ test} \\
\{\text{gauche.sequence}\} \text{ Groups consecutive elements in a sequence seq which have the common key value. A key value of an element is obtained by applying the procedure key to the element; the default procedure is identity. For each element in seq, key is applied exactly once. The equal-ness of keys are compared by test procedure, whose default is eqv?.}
\]

\[
\text{(group-sequence '}(1 \ 1 \ 1 \ 2 \ 3 \ 4 \ 4 \ 2 \ 2 \ 3 \ 1 \ 1 \ 3)) \\
\Rightarrow \ ((1 \ 1 \ 1) \ (2) \ (3) \ (4 \ 4) \ (2 \ 2) \ (3) \ (1 \ 1) \ (3))
\]
\[
\text{(group-sequence '}(1 \ 1 \ 1 \ 2 \ 3 \ 4 \ 4 \ 2 \ 2 \ 3 \ 1 \ 1 \ 3) \\
:key \ (\text{cut modulo} \ <> \ 2))) \\
\Rightarrow \ ((1 \ 1 \ 1) \ (2) \ (3) \ (4 \ 4 \ 2 \ 2) \ (3 \ 1 \ 1 \ 3))
\]
\[
\text{(group-sequence '}("a" "a" "b" "b" "c" "d" "d") \\
:test \ \text{string=?}) \\
\Rightarrow \ (("a" "a") \ ("b" "b") \ ("c") \ ("d" "d"))
\]
\[
\text{(group-sequence "aabbccd" \\
:test \ char=?)} \\
\Rightarrow \ ((\#\a \ #\a) \ (\#\b \ #\b) \ (\#\c) \ (\#\d \ #\d))
\]

This method is similar to Haskell’s `group`. If you want to group elements that are not adjacent, use `group-collection` (see Section 9.6.2 [Selection and searching in collection], page 408).

If you simply need to reduce each group for one instance, that is, removing adjacent duplicated elements, you can use `delete-neighbor-dups` below.

\[
\text{group-contiguous-sequence \ seq \ :key \ key \ next \ test \ squeeze} \\
\{\text{gauche.sequence}\} \text{ Group contiguous elements in seq.}
\]

\[
\text{(group-contiguous-sequence '}(1 \ 2 \ 3 \ 4 \ 7 \ 8 \ 9 \ 11 \ 13 \ 14 \ 16)) \\
\Rightarrow \ ((1 \ 2 \ 3 \ 4) \ (7 \ 8 \ 9) \ (11) \ (13 \ 14) \ (16))
\]

If the keyword argument `squeeze` is true, each subsequence is represented with its first and last elements, except when the subsequence has only one element.

\[
\text{(group-contiguous-sequence '}(1 \ 2 \ 3 \ 4 \ 7 \ 8 \ 9 \ 11 \ 13 \ 14 \ 16) :squeeze \ #t) \\
\Rightarrow \ ((1 \ 4) \ (7 \ 9) \ (11) \ (13 \ 14) \ (16))
\]

The keyword argument `key` must be a procedure taking one argument, and it is applied to every element in the sequence once, to construct the result. Its default is `identity`.

The keyword argument `next` must be a procedure taking one argument, which is the key value (whatever `key` procedure returns) and must return the “next” key value. Its default is `(^n (+ n 1))`.

The `test` argument must be a procedure taking two argument and used to compare two key values. Its default is `eqv?`.

\[
\text{(group-contiguous-sequence "AbCdFgH" \\
:key \ \text{char-upcase} \ :next \ ((\text{c} \ (\text{integer->char} (+ 1 \ (\text{char->integer} c))))))} \\
\Rightarrow \ ((\#\A \ #\B \ #\C \ #\D) \ (#\F \ #\G \ #\H))
\]
**delete-neighbor-dups** `seq :key key test start end`  
{gauche.sequence}  
Returns a sequence of the same type as `seq`, in which elements in `seq` are included in the original order, except duplicate adjacent elements. The type of `seq` must have a builder.

```scheme
(delete-neighbor-dups '(1 1 1 2 3 4 4 2 2 3 1 1 3))
⇒ (1 2 3 4 2 3 1 3)
```

```scheme
(delete-neighbor-dups '#(1 1 1 2 3 4 4 2 2 3 1 1 3))
⇒ #(1 2 3 4 2 3 1 3)
```

```scheme
(delete-neighbor-dups "1112344223113")
⇒ "12342313"
```

Elements are compared with `eqv?` by default. You can pass alternative procedure to `test` keyword argument; it is always called as `(test x y)`, where `x` and `y` are the contiguous elements in `seq`. If elements are compared equal, the first one is kept:

```scheme
(delete-neighbor-dups "AaaAbBBbCCcc" :test char-ci=?)
⇒ "AbC"
```

If `key` is provided, it must be a procedure that takes one arguments. It is applied to each element of `seq` at most once, and each resulting value is used for the comparison instead of elements themselves.

```scheme
(delete-neighbor-dups
  '((1 . "a") (1 . "b") (2 . "c") (2 . "d"))
  :key car)
⇒ ((1 . "a") (2 . "c"))
```

The `start` and `end` arguments specify indexes in the `seq` to limit the range to look at. Where `start` is inclusive, `end` is exclusive.

```scheme
(delete-neighbor-dups "1112344223113" :start 3 :end 9)
⇒ "2342"
```

**delete-neighbor-dups!** `seq :key key test start end`  
{gauche.sequence}  
Scan `seq` from left to right, dropping consecutive duplicated elements. The result is stored into `seq`, packed to left. Note `seq` must be modifiable by index, i.e. modifier method must be defined. The rest of `seq` will be untouched. Returns the next index after the last modified entry.

```scheme
(let1 v (vector 1 1 2 2 3 3 2 2 4 4)
  (list (delete-neighbor-dups! v)
    v))
⇒ (5 #(1 2 3 2 4 3 2 2 4 4))
```

The semantics of keyword arguments `key`, `test`, `start` and `end` are the same as `delete-neighbor-dups`.

```scheme
(let1 v (vector 1 1 2 2 3 3 2 2 4 4)
  (list (delete-neighbor-dups! v :start 2)
    v))
⇒ (6 #(1 1 2 3 2 4 2 2 4 4))
```

Note: This method works on any sequence with modifier method, but it’s not necessarily more efficient than `delete-neighbor-dups`, which creates a new sequence. If `seq` is a list or a string, each modification by index takes O(n) time (for a string even it costs O(n) extra storage), so the total cost is O(n^2), whereas `delete-neighbor-dups` needs O(n) time and storage. This works best for vectors and alike, with which it doesn’t cost extra allocation and runs in O(n) time.
delete-neighbor-dups-squeeze! seq :key key test start end  [Generic function]  
{gauche.sequence} Operates like delete-neighbor-dups but reuses storage of seq, which will be resized by dropping duplicated elements. Returns the sequence after dupes are removed.

Not all sequences are resizable, so this method won’t be defined for such sequences. The gauche.sequence module only provides this method for <list>, in which dropping the middle of the sequence is very efficient as it is just a single set-cdr!

(delete-neighbor-dups-squeeze! (list 1 1 1 2 2 2 3 3 3))  ⇒  (1 2 3)

(delete-neighbor-dups-squeeze! (list 1 1 1 2 2 2 3 3 3 4 4) :start 3 :end 9)  ⇒  (2 3)

The semantics of keyword arguments key, test, start and end are the same as delete-neighbor-dups.

Prefix

common-prefix (a <sequence>) (b <sequence>) :key key test  [Generic function]  
{gauche.sequence} Returns a new sequence of the same type of a which contains the common prefix of sequences a and b. The types of a and b doesn’t need to match. The type of a must have a builder.

For each corresponding element in a and b, the key procedure is applied (default identity), then compared with test procedure (default eqv?).

(common-prefix '(a b c d e) '(a b c e f))  ⇒  (a b c)

(common-prefix "abcef" '#(\a \b \c \d \e))  ⇒  "abc"

For strings, SRFI-13 has a specific function with related feature: string-prefix-length (see Section 11.6.6 [SRFI-13 String prefixes & suffixes], page 696).

common-prefix-to (class <class>) (a <sequence>) (b <sequence>)  [Generic function]  
{gauche.sequence} Returns a new sequence of the type class which contains the common prefix of sequences a and b. The types of a and b doesn’t need to match, and neither needs to have a builder. The class must be a sequence class with a builder.

The meanings of keyword arguments are the same as common-prefix.

(common-prefix-to <list> "abcde" "ABCEF" :test char-ci=?)  ⇒  '('#\a '#\b '#\c)

Permutation and shuffling

permute (src <sequence>) (permuter <sequence>) :optional fallback  [Generic function]  
{gauche.sequence} Returns a newly created sequence of the same type as src, in which the elements are permuted from src according to permuter.

Permuter is a sequence of exact integers. When the k-th element of permuter is i, the k-th element of the result is (ref src i). Therefore, the size of the result sequence is the same as the size of permuter. Permuter can be any kind of sequence, unrelated to the type of src.

It is allowed that the same index i can appear more than once in permuter.

(permute '(a b c d) '(3 2 0 1))  ⇒  (d c a b)
(permute '(a b c d) '(0 2))  ⇒  (a c)
(permute '(a b c d) '(0 0 1 1 2 2))  ⇒  (a a b b c c)

If an integer in *permuter* is out of the valid range as the index of *src*, then an error is signaled unless *fallback* is given. If *fallback* is given, what value is used depends on the result of (ref *src* i *fallback*)—which usually returns *fallback* for the out-of-range index i.

(permute '#(a b c) '(3 2 1 0) 'foo)  ⇒  #(foo c b a)

(permute "!,HWdelor" #(2 5 6 7 1 -1 3 7 8 6 4 0) #\space)  ⇒  "Hello, World!"

**permute-to** (class <class>) (src <sequence>) (permuter <sequence>) :optional **fallback**
{gauche.sequence} Like *permute*, but the result will be an instance of the given *class* instead of the class of *src*.

(permute-to <string> '#\a #\b #\c #\d #\r)  ⇒  "abracadabra"

**permute!** (src <sequence>) (permuter <sequence>) :optional **fallback**
{gauche.sequence} Also like *permute*, but the result is stored back to *src*. *Src* must be a mutable sequence, and the length of *src* and *permuter* must be the same.

**shuffle** (src <sequence>) :optional **random-source**
{gauche.sequence} Returns a new sequence of the same type and size as *src*, in which elements are randomly permuted.

(shuffle '(a b c d e))  ⇒  (e b d c a)
(shuffle "abcde")  ⇒  "baced"

This generic function uses SRFI-27 (see Section 11.8 [Sources of random bits], page 706). By default it uses default-random-source, but you can pass an alternative random source by the optional argument.

**shuffle-to** (class <class>) (src <sequence>) :optional **random-source**
{gauche.sequence} Like *shuffle*, except that the result will be an instance of *class* instead of the class of *src*.

**shuffle!** (src <sequence>) :optional **random-source**
{gauche.sequence} Like *shuffle*, but the result is stored back to *src*. *Src* must be a mutable sequence.

### 9.31.5 Implementing sequence

#### 9.32 gauche.syslog - Syslog

**gauche.syslog**

This module provides syslog(3) system logger interface.

For the common applications, you might find *gauche.logger* module easier to use (see Section 9.17 [User-level logging], page 458). This module is for those who need direct access to the syslog API.

The procedures are only defined if the underlying system supports them.
sys-openlog ident option facility

[Function]
[POSIX] {gauche.syslog} Opens a connection to the system logger. A string argument
ident is used for the prefix of the log, and usually is the program name. Option is an integer
flag to control the behavior of logging, and facility is an integer that specify the type of the
program.

The flag for option can be composed by logior-ing one or more of the following integer con-
stants: LOG_CONS, LOG_NDELAY, LOG_NOWAIT, LOG_ODELAY, LOG_PERROR and LOG_PID. (Some
of the constants may not be defined if the underlying system doesn’t support them).

The facility argument can be one of the following integer constants: LOG_AUTH, LOG_
AUTHPRIV, LOG_CRON, LOG_DAEMON, LOG_FTP, LOG_KERN, LOG_LOCAL0 through LOG_LOCAL7,
LOG_LPR, LOG_MAIL, LOG_NEWS, LOG_SYSLOG, LOG_USER and LOG_UUCP. (Some of the con-
stants may not be defined if the underlying system doesn’t support them).

See your system’s manpage of openlog(3) for detail description about these constants.

sys-syslog priority message

[Function]
[POSIX] {gauche.syslog} Log the string message. Unlike syslog(3), this procedure doesn’t
do formatting—you can use format (see Section 6.21.8 [Output], page 275) to create a for-
matted message, or use higher-level routine log-format (see Section 9.17 [User-level logging],
page 458).

An integer argument priority can be composed by logior-ing one of the facility constants
described above and the level constants: LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_
WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG.

sys-closelog

[Function]
[POSIX] {gauche.syslog} Closes the connection to the logging system.

sys-setlogmask mask

[Function]
[POSIX] {gauche.syslog} Sets the process’s log priority mask that determines which calls
to sys-syslog may be logged. An priority mask can be composed by logior-ing bitmasks
corresponding to the level argument of sys-syslog. You can use sys-logmask below to
obtain a bitmask from the level.

sys-logmask level

[Function]
[POSIX] {gauche.syslog} Returns an integer bitmask for sys-setlogmask from the log
level level.

9.33 gauche.termios - Terminal control

gauche.termios

[Module]
This module provides procedures to control terminals. On Unix platforms, the low-level API
provides POSIX termios interface as the module name suggests. This module also provides
pseudo tty interface, if the system supports it.

On Windows native platforms, POSIX termios interface is not available. It is too different
from Windows console API to provide a meaningful emulation. The low-level Windows
console API is available in the os.windows module (see Section 12.39 [Windows support],
page 898). You can still use high-level terminal control procedures in this module.

9.33.1 Posix termios interface

These procedures are available when the feature identifier gauche.os.windows is not defined.
See cond-expand in Section 4.12 [Feature conditional], page 75, for how to switch code using
feature identifiers.
Builtin Class <sys-termios>

POSIX termios(7) structure.

- **iflag**: Instance Variable of <sys-termios>
- **oflag**: Instance Variable of <sys-termios>
- **cflag**: Instance Variable of <sys-termios>
- **lflag**: Instance Variable of <sys-termios>
- **cc**: Instance Variable of <sys-termios>

The slots `iflag`, `oflag`, `cflag` and `lflag` contain non-negative integers representing bit-masks.

The slot `cc` contains a copy of `c_cc` array of `struct termios`, as an u8vector (see Section 6.13.2 [Uniform vectors], page 198, for the details about u8vector). Since `cc` slot is a copy of the internal structure, you have to set! an u8vector to the slot explicitly to make changes to the `c_cc` array.

Throughout this section, argument `port-or-fd` refers to either a port object or a small integer representing system’s file descriptor. If `port` is not associated to the system terminal, an error is signaled. (You can check if `port` has an associated terminal by `sys-isatty?`. see Section 6.24.4.5 [Other file operations], page 304).

**sys-tcgetattr** `port-or-fd`

{gauche.termios} Returns terminal parameters in a <sys-termios> object, associated to `port-or-fd`.

**sys-tcsetattr** `port-or-fd when termios`

{gauche.termios} Sets terminal parameters associated to `port-or-fd` by `termios`, which must be an instance of <sys-termios>.

An integer argument `when` specifies when the changes take effect. Three variables are predefined for the argument:

- **TCSANOW** The change is reflected immediately.
- **TCSADRAIN** The change is reflected after all pending output is flushed.
- **TCSAFLUSH** The change is reflected after all pending output is flushed, and all pending input is discarded.

**sys-tcsendbreak** `port-or-fd duration`

{gauche.termios} Transmits a zero stream for the specified duration to the terminal associated to `port-or-fd`. The unit of duration depends on the system; see man tcsendbreak(3) of your system for details.

**sys-tcdrain** `port-or-fd`

{gauche.termios} Waits until all output written to `port-or-fd` is transmitted.

**sys-tcflush** `port-or-fd queue`

{gauche.termios} Discards data in the buffer of `port-or-fd`, specified by `queue`, which may be one of the following values.

- **TCIFLUSH** Discards data received but not read.
- **TCOFLUSH** Discards data written but not transmitted.
- **TCIOFLUSH** Do both TCIFLUSH and TCOFLUSH action.
Chapter 9: Library modules - Gauche extensions

sys-tcflow port-or-fd action
{gauche.termios} Controls data flow of port-or-fd by action, which may be one of the following values:
TCOFF  Suspends output transmission.
TCON  Restarts output transmission.
TCIOFF  Transmits a STOP character to make the terminal device stop transmitting data to the system.
TCION  Transmits a START character to make the terminal device resume transmitting data to the system.

sys-tcgetpgrp port-or-fd
{gauche.termios} Returns process group ID of the terminal associated to port-or-fd.

sys-tcsetpgrp port-or-fd pgrp
{gauche.termios} Sets process group ID of the terminal associated to port-or-fd to pgrp.

sys-cfgetispeed termios
{gauche.termios} Gets/sets input/output speed (baud rate) parameter stored in termios object. Speed is represented by the following predefined numbers:
B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400.
Some system may support higher baud rate, such as B57600, B115200 or B230400. You can use symbol-bound? to check these options are defined. B0 is used to terminate the connection.

sys-openpty :optional term
{gauche.termios} Opens a pair of pseudo ttys, one for master and the other for slave, then returns two integers which are their file descriptors. An optional argument term must be, if passed, a <sys-termios> object; it sets the slave pty’s parameters.
You can use open-input-fd-port and/or open-output-fd-port to create a port around the returned file descriptor (see Section 6.21.4 [File ports], page 264). To obtain pseudo tty’s name, use sys-ttynname (see Section 6.24.4.5 [Other file operations], page 304).
This function is available only if the system supports openpty(3).

sys-forkpty :optional term
{gauche.termios} Opens a pair of pseudo ttys, one for master and the other for slave, sets the slave pty suitable for login terminal, then fork(2).
Returns two integers; the first value is a child pid for the parent process, and 0 for the child process. The second value is a file descriptor of the master pty.
An optional argument term must be, if passed, a <sys-termios> object; it sets the slave pty’s parameters.
This function is available only if the system supports forkpty(3).
Note: sys-forkpty has the same MT hazard as sys-fork (see Section 6.24.10 [Process management], page 318, for details). If you’re running multiple threads, use sys-forkpty-and-exec below.

sys-forkpty-and-exec command args :key iomap term sigmask
{gauche.termios} Does sys-forkpty, and lets the child process immediately execs the specified command with arguments args. This function doesn’t have the hazard in multi-thread environment.
The meanings of arguments command, args, iomap and sigmask are the same as sys-exec (see Section 6.24.10 [Process management], page 318). If the keyword argument term is given, it is used to initialize the slave pty.

### 9.33.2 Common high-level terminal control

**without-echoing** *iport proc*  
{gauche.termios} If *iport* is an input port connected to a terminal, sets the terminal mode non-echoing and call **proc** with *iport* as an argument. Before returning from without-echoing, or throwing an error, the terminal mode is reset to the original state when this procedure is called. The procedure returns whatever value(s) **proc** returns.

You can also pass #f to *iport*. In that case, this procedure tries to open a console (/dev/tty on Unix, CON on Windows) and set the console mode, then calls **proc** with the opened input port. An error is thrown if the procedure can not open a console.

If *iport* is other than above, this procedure simply calls **proc** with *iport*. This allows the caller to read password from redirected input, for example.

Note: Because of an implementation issue, on Windows native platforms this procedure always changes console mode of the standard input handle when *iport* is either #f or a terminal input port.

**has-windows-console?**  
{gauche.termios} Returns #t iff the running Gauche is Windows-native and the process has attached console. On Unix platforms this procedure always returns #f.

The reason that cond-expand isn’t enough is that on Windows the program may start without console, but you can attach console afterwards. See Section 12.39.2 [Windows console API], page 898, for the details.

### 9.34 gauche.test - Unit Testing

**gauche.test**  
{Module} Defines a set of functions to write test scripts. A test script will look like this:

```lisp
(use gauche.test)
(test-start "my feature")
(load "my-feature") ; load your program
(import my-feature) ; if your program defines a module.
(test-module 'my-feature) ; tests consistency in your module.
(test-section "feature group 1")
(test "feature 1-1" EXPECT (lambda () TEST-BODY))
(test "feature 1-2" EXPECT (lambda () TEST-BODY))
...
(test-section "feature group 2")
(define test-data ...)
(test "feature 2-1" EXPECT (lambda () TEST-BODY))
(test "feature 2-2" (test-error) (lambda () TEST-THAT-SIGNALS-ERROR))
...
(test-end :exit-on-failure #t)
```

With this convention, you can run test both interactively or in batch. To run a test interactively, just load the file and it reports a result of each test, as well as the summary of failed
test at the end. To run a test in batch, it is convenient to redirect the stdout to some file
If stdout is redirected to other than tty, all the verbose logs will go there, and only a small
amount of messages go to stderr.

It is recommended to have a "check" target always in Makefile of your module/program, so
that the user of your program can run a test easily. The rule may look like this:

   check :
       gosh my-feature-test.scm > test.log

For the portable programs, there are a couple of srfis that provide testing frameworks (see
Section 11.19 [Lightweight testing], page 725, and see Section 11.15 [A Scheme API for test
suites], page 720). In Gauche, when you can use these srfis with gauche.test, srfi’s tests
work as a part of Gauche’s tests.

9.34.1 Structuring a test file

test-start module-name [Function]
   {gauche.test} Initializes internal state and prints a log header. This should be called before
   any tests. Module-name is used only for logging purpose.

test-section section-name [Function]
   {gauche.test} Marks beginning of the group of tests. This is just for logging.

test-log fmtstr args . . . [Function]
   {gauche.test} This is also just for logging. Creates a formatted string with fmtstr and args
   just like format, then write it to the current output port, with prefix ;; and newline at the
   end.

   With the typical Makefile settings, where you redirect stdout of test scripts to a log file, the
   message only goes to the log file.

   Using this, you can dump information that can’t be automatically tested but may be useful for
troubleshooting. For example, you get a mysterious test failure reports you can’t reproduce
on your machine, and suspect some aspects of the running systems may unpredictably affect
the test result. You can put test-log in the test code to dump such parameters, and ask
the reporter to run the test again and analyze the log.

test-end :key exit-on-failure [Function]
   {gauche.test} Prints out list of failed tests. If exit-on-failure is #f or omitted, this procedure
   returns the number of failed tests.

   Otherwise, this function terminates the gosh process by exit. If a fixnum is given to
   exit-on-failure it becomes the process’s exit status; if other true value is given, the exit
   status will be 1.

test-record-file file [Function]
   {gauche.test} Suppose you have several test scripts. Normally you run them as a group
and what you want to know is a concise summary of the whole results, instead of each result
of individual test files.

   A test record file is an auxiliary file used to gather summary of the result. It holds a one-line
summary of tests like this:

   Total: 9939 tests, 9939 passed, 0 failed, 0 aborted.

   When a test record file exists, test-start reads and parses it, and remembers the numbers.
   Then test-end adds the count of the results and writes them back to the same test record
file.
If you write the `check` target in your makefile as follows, you will get the final one-line summary every time you run `make check`, assuming that `test1.scm`, `test2.scm`, and `test3.scm` all have `(test-record-file "test.record")` before a call to `test-start`.

```makefile
check:
  @rm -f test.record test.log
gosh test1.scm >> test.log
gosh test2.scm >> test.log
gosh test3.scm >> test.log
@cat test.record
```

Note that to make `test-record-file` work, it must be placed before the call to `test-start`. Alternatively, you can use the environment variable `GAUCHE_TEST_RECORD_FILE` to specify the test record file.

`GAUCHE_TEST_RECORD_FILE`  
[Environment Variable]  
If this environment variable is set when the test script is run, its value is used as the name of the test record file. If the test script calls `test-record-file`, it takes precedence and this environment variable is ignored.

`test-summary-check`  
[Function]  
{gauche.test} If the test record file is set (either by `test-record-file` or the environment variable `GAUCHE_TEST_RECORD_FILE`), read it, and then exit with status 1 if the record has nonzero failure count and/or nonzero abort count. If the test record file isn’t set, this procedure does nothing. This is useful when you have multiple test scripts and you want to let `make` fail if any of tests fails, but not before all test script is run. If you make every test script use `:exit-on-failure` of `test-end`, then `make` stops immediately after the script that fails. Instead, you avoid using `:exit-on-failure`, but use the test record file and for the last thing you can call this function:

```makefile
check:
  rm -f $GAUCHE_TEST_RECORD_FILE test.log
gosh test1.scm >> test.log
gosh test2.scm >> test.log
cat $GAUCHE_TEST_RECORD_FILE /dev/null
gosh -ugauche.test -Etest-summary-check -Eexit
```

By this, `make` will run all the test script no matter how many of them fails (since `gosh` exits with status 0), but detect an error since the last line of `gosh` call exits with status 1 if there has been any failure.

### 9.34.2 Individual test

Most test frameworks have various test procedures such as `test-assert`, `test-equal`, etc., depending on what you want to test. We have only one, `test`, and its thin wrapper `test*`. It takes a thunk (or an expression) to run, the expected result, and an optional check predicate to compare the expected result against the actual result. Various conditions, such as testing the actual result is a true value, or testing the expression raising a specific error, can be handled by the check predicate. The default check predicate handles typical cases:

- Compare the expected result with the actual result.

  ```scheme
  (test* "one plus one equals two"
       2
       (+ 1 1))
  ```
• Assert the expression returns true value (i.e. not #f).
  (test* "'any' returns non-false value"
      (test-truthy)
      (any integer? '(1.2 3/4 5)))
• Check if the actual result is one of possible values.
  (test* "get random integer between 0 and 5"
        (test-one-of 0 1 2 3 4 5)
        (random-integer 6))
• Check if the expression raises an expected error.
  (test* "expects read error"
        (test-error <read-error>)
        (read-from-string "(a .)"))

With this way, it is easier to parameterize tests, e.g. loop over a list of expected result and test thunks.

test* name expected expr :optional check hook
{gauche.test} A convenience macro that wraps expr by lambda.
  (test* name expected expr) ≡ (test name expected (lambda () expr))

test name expected thunk :optional check report hook
{gauche.test} Calls thunk, and checks its result fits expected using a procedure check, which is called as follows:
  (check expected result-of-thunk)

It should return #t if the given result agrees with the expected value, or #f otherwise. The default check procedure is test-check, explained below. It compares expected and result-of-thunk with equal?, except when expected is some of special case test objects. (See “testing ambiguous results” and “testing abnormal cases” paragraphs below for this special treatment.)

One typical usage of the custom check procedure is to compare inexact numbers tolerating small error.
  (test "test 1" (/ 3.141592653589 4)
       (lambda () (atan 1))
       (lambda (expected result)
            (< (abs (- expected result)) 1.0e-10)))

Name is a name of the test, for the logging purpose.

When thunk signals an uncaptured error, it is caught and yields a special error object <test-error>. You can check it with another error object created by test-error function to see if it is an expected type of error. See the entry of test-error below for the details.

The report optional argument must be an #f or a procedure that takes three arguments. If it is a procedure, it is called after check returns false (but before hook is called). The first argument is name, the second argument is expected, and the third argument is either the result of thunk, or a <test-error> object when thunk raises an error. The default is test-report-failure procedure, which simply uses write to display the result of thunk or a <test-error> object. By passing your own procedure, you can customize the message to be printed when the test is failed.

Finally, the hook optional argument must be an #f or a procedure that takes four arguments. If it is a procedure, it is called after the test procedure finishes the test. The first argument is
a symbol either pass or fail, the second argument is name, the third argument is expected,
and the fourth argument is either the result of thunk, or a <test-error> object when thunk
raises an error. The return value of hook is ignored.

It is mainly for libraries that wrap gauche.test and wants to do its own bookkeeping.

Note: In 0.9.10, we didn’t have report argument. Instead of adding report to the last optional
argument, we made it the second and shifted hook, for hook argument will rarely be used. To
keep the backward compatibility, we recognize if 4-argument procedure is passed to a report
argument we treat it as hook, with warning. This compatibility feature will be removed in
future releases.

test-check expected actual :optional equal
{gauche.test} The default procedure test and test* use to check the result of the test
expression conforms the expected value. By default, test-check just compares expected and
actual with a procedure equal, which is defaulted to equal?. test-check behaves differently
if expected is one of special test objects described below.

test-report-failure name expected actual
{gauche.test} The default procedure to report test failure. It just writes actual with write.
You can customize the failure report by passing your reporting procedure to report argument
to test and test*. See test-report-failure-diff below, for example.

Testing ambiguous results

test-one-of choice . . .
{gauche.test} Sometimes the result of test expression depends on various external environ-
ment, and you cannot put an exact expected value. This procedure supports to write such
tests conveniently.

Returns a special object representing either one of the choices. The default check procedure,
test-check, recognizes the object when it is passed in the expected argument, and returns
true if any one of choice . . . passes the check against the result.

For example, the following test passes if proc returns either 1 or 2.

(test* "proc returns either 1 or 2" (test-one-of 1 2) (proc))

Note that test-check compares the actual result against each of choices by test-check
itself, that is:

(test-check (test-one-of choice ...) result equal)
≡ (or (test-check choice result equal) ...)

This, if you want to compare each choice with customized equivalence procedure, pass
test-check with a specialized equivalence procedure as the check procedure. The following
example compares each choice and the result case-insensitively:

(test* "Using one-of with case insensitive comparison"
 (test-one-of "abc" "def")
 "Abc"
 (cut test-check <> <> string-ci=?))

test-none-of choice . . .
{gauche.test} Similar to test-one-of, but creates a special object representing none of
the choices. The test succeeds if the test expression evaluates to a value that don’t match any
of choices.

Note: If you want to compare inexact numeric result, you can use approx=? (see Section 6.3.3
[Numerical comparison], page 125).
test-truthy
{gauche.test} Returns a special object that expects a true value, e.g. anything but #f. Use it to assert the test expression returns a true value but not necessarily #t.

Testing abnormal cases

test-error :optional (condition-type <error>) (message #f)
{gauche.test} Returns a new <test-error> object that matches with other <test-error> object with the given condition-type.

The test-check procedure treats <test-error> objects specially. When err-expected and err-actual are <test-error> objects, (test-check err-expected err-actual) returns #t if err-expected’s condition type is the same as or supertype of err-actual’s.

For example, if you want to test a call to foo raises an <io-error> (or its condition subtype), you can write as the following example:

(test "see if foo raises <io-error>" (test-error <io-error>) (foo))

Another optional argument message can be used to check if the raised error has a message of expected pattern. The argument may be a string, a regexp or #f (default). If it is a string, test-check checks if the message of the raised error exactly match the string. If it is a regexp, test-check checks the message of the raised error matches that regexp. If it is #f, the message is not checked.

*test-error*
{gauche.test} Deprecated. Bounded to an instance of <test-error> with condition type <error>. This is only provided for the backward compatibility; new code should use test-error procedure above.

*test-report-error*
{gauche.test} If this variable is true, the test routine prints stack trace to the current error port when it captures an error. It is useful when you got an unexpected test-error object and want to check out where the error is occurring.

This variable is initialized by the environment variable GAUCHE_TEST_REPORT_ERROR when the gauche.test module is loaded. For example, you can use the environment variable to check out an unexpected error from your test script as follows (the value of the environment variable doesn’t matter).

env GAUCHE_TEST_REPORT_ERROR=1 gosh mytest.scm

9.34.3 Incorporating external tests

Sometimes you implement an existing specification that comes with tests. If tests are written in R7RS Scheme, you can run it by itself; however, you might want to run it as a part of larger test suite managed by gauche.test. By doing so, you can get consolidated test result reports, for Gauche integrates SRFI test frameworks such as srfi.64 and srfi.78.

If the test file is written in R7RS format, however, you may not be able to simply include the test file into Gauche’s test suite. As R7RS import differs from Gauche’s, we have some black magic to switch R7RS/Gauche namespace based on whether the first form in the file is R7RS import form or not (see Section 10.1.2 [Three forms of import], page 582). If you include R7RS program into Gauche’s program, the import in the R7RS code is interpreted as Gauche’s and doesn’t work. The macro test-include-r7 can be used instead.

NB: If the external test is written for Chibi Scheme, you can use chibi-test instead (see Section 12.3 [Running Chibi-scheme test suite], page 814).
test-include-r7 path [exclude-clause]  [Macro]

{gauche.test} Like include form (see Section 4.11 [Inclusions], page 74), expands the content of the file specified by path in place of this form. If path is relative, it is relative from the includer.

The code is included in the environment where import is bound to R7RS’s, so that a test script written for R7RS can be included as is.

A recommended usage is to create a submodule in Gauche’s test script:

```
(use gauche.test)
(test-start ...)
:
(test-section "xxx-tests")
(define-module xxx-tests
  (use gauche.test)
  (test-include-r7 "xxx-tests"))
```

Sometimes the external script refers to a library that’s not correspond to what Gauche provides (e.g. tests/include/srfi-222-tests.scm imports (compounds) library, but Gauche provides it as srfi.222.) You can list such libraries to exclude clause so that import won’t load it:

```
(define-module srfi-222-tests
  (use gauche.test)
  (use srfi.222)
  (test-include-r7 "include/srfi-222-tests" (exclude (compounds))))
```

The format of exclude-clause is as follows:

```
<exclude-clause> : (exclude r7rs-library-name ...)
```

### 9.34.4 Better test failure reporting

As described in test entry above, you can customize how the failure is reported by passing the optional report argument to test and test*. One of useful customizations is to show the difference between multi-line text. It’s such a useful tool so we provide a report procedure.

Here’s a contrived example. We pass test-report-failure-diff as a report procedure (and test-check-diff for check procedure, which we’ll explain later). Expected text is given as a list of lines, while the actual result is a single string; Both test-report-failure-diff and test-check-diff procedures canonicalize expected and actual result into a single multi-line string, so you can give them in whichever ways that’s convenient for you.

```
(test* "Beatrice"
  ;; expected
  '("What fire is in mine ears? Can this be true?"
    "Stand I condemned for pride and scorn so much?"
    "Contempt, farewell, and maiden pride, adieu!"
    "No glory lives behind the back of such."
  )
  ;; actual
  "What fire is in mine ears? Can this be true?
  Stand I condemn’d for pride and scorn so much?
  Contempt, farewell! and maiden pride, adieu!
  No glory lives behind the back of such.
"
(test-check-diff ; check
(test-report-failure-diff) ; report
⇒ Reports:
ERROR: GOT diffs:
--- expected
+++ actual
@@ -1,4 +1,4 @@
What fire is in mine ears? Can this be true?
-Stand I condemned for pride and scorn so much?
-Contempt, farewell, and maiden pride, adieu!
+Stand I condemn’d for pride and scorn so much?
+Contempt, farewell! and maiden pride, adieu!
No glory lives behind the back of such.

As you see, the result is reported in a unified diff format (see Section 12.65 [Calculate difference of text streams], page 991) so that you can spot the difference easily.

test-check-diff expected actual :optional equal
{gauche.test} An alternative check procedure you can pass into check argument of test procedure / test* macro.
Before comparing expected and actual, it performs the following operations on each of expected and actual:

- If it is a list of strings, join them with \n (with suffix syntax, so the last line is also appended with \n).
- If it is a form (content-of <string>), then <string> is taken as a filename and the content of the file is used as a string. If the filename is relative, it is relative to the current loading file. If named file doesn’t exist, an empty string is used.
- If it is a string or other object, it is used as is.

Then the two arguments are compared using equal, which is defaulted to equal?.

test-report-failure-diff msg expected actual
{gauche.test} An alternative failure report procedure you can pass into report argument of test procedure / test* macro.
The expected and actual arguments are converted in the same way as test-check-diff; that is, if it is a list of strings (lines) or a form (content-of <filename>), it is converted to a single string.
Then the difference of the two is reported in a unified diff format (using diff-report/unified. See Section 12.65 [Calculate difference of text streams], page 991).
If either expected or actual is not convertible to a single string, the result is reported in the same way as the standard test-report-failure.

Note: This procedure is called twice, once when the test is failed, and again from test-end to report the summary of discrepancy. If you pass (content-of <filename>) form, you have to make sure the named file exists until test-end returns. This is tricky if you generate text into a temporary file during a single test. In general, (content-of <filename>) form is useful in the expected argument, where you can specify the prepared file.

test*/diff name expected expr
{gauche.test} This is a convenience version of test*, using test-check-diff and test-report-failure-diff as check and report procedures, respectively.
(test*/diff name expected expr)
≡
(test* name expected expr test-check-diff test-report-failure-diff)
9.34.5 Quasi-static checks

Scheme is dynamically typed, which is convenient for incremental and experimental development on REPL, but it tends to delay error detection until the code is actually run. It is very annoying that you run your program for a while only to see it barfs on simple typo of variable name.

Gauche addresses this issue by checking certain types of errors at the test phase. It isn’t purely a static check (we need to load a module or a script, which evaluates toplevel expressions), nor exhaustive (we can’t catch inconsistencies that span over multiple modules or about information that can be added at runtime). Nevertheless it can often catch some common mistakes, such as incorrect variable names or calling procedures with wrong number of arguments.

The two procedures, `test-module` and `test-script`, load the named module and the script files respectively (which compiles the Scheme code to VM instructions), then scan the compiled VM code to perform the following tests:

1. See if the global variables referenced within functions are all defined (either in the module, or in one of imported modules).
2. If a global variable is used as a function, see if the number of arguments given to it is consistent to the actual function.
3. See if the symbols set as autoload in the code can be resolved.
4. While testing module, see if the symbols declared in the export list are actually defined.

The check is somewhat heuristic and we may miss some errors and/or can have false positives. For false positives, you can enumerate symbols to be excluded from the test.

**test-module module :key allow-undefined bypass-arity-check**  
{gauche.test} Loads the module and runs the quasi-static consistency check. *Module* must be a symbol module name or a module.

Sometimes you have a global variable that may not be defined depending on compiler options or platforms, and you check its existence at runtime before using it. The undefined variable reference check by `test-module` doesn’t follow such logic, and reports an error whenever it finds your code referring to undefined variable. In such case, you can give a list of symbols to the *allow-undefined* keyword argument; the test will excludes them from the check.

The arity check may also raise false positives, if the module count on a behavior of global procedures that will be modified after the module is loaded (e.g. a method with different number of arguments can be added to a generic function after the module is loaded, which would make the code valid.) If you know you’re doing right thing and need to suppress the false positives, pass a list of names of the functions to *bypass-arity-check* keyword arguments.

**test-script filename :key allow-undefined bypass-arity-check compile-only**  
{gauche.test} Loads the script named by *filename* into a fresh anonymous module and runs the quasi-static consistency check. *Filename* must be a string that names the script file.

The meaning of keyword arguments is the same as `test-module`.

Note that the toplevel forms in *filename* are evaluated, so scripts that relies on the actions of toplevel forms could cause unwanted side-effects. This check works best for the scripts written in SRFI-22 convention, that is, calling actions from *main* procedure instead of toplevel forms. R7RS scripts relies on actions in toplevel forms and can’t be tested with this procedure.

Scripts that relies on being loaded into *user* module also won’t work well with this check, which loads the forms into anonymous module.

If you need to test a script with toplevel side-effecting forms and you can’t change it, you may want to pass true value to the *compile-only* keyword argument. Then `test-script`
just compiles each toplevel form before running static checking, instead of loading (which not only compiles but executes each of toplevel forms).

9.35 gauche.threads - Threads

If enabled at compilation time, Gauche can use threads built on top of either POSIX threads (pthreads) or Windows threads.

`gauche.threads` [Module]

Provides thread API. You can ‘use’ this module regardless whether the thread support is compiled in or not; if threads are not supported, many thread-related procedures simply signals a "not supported" error.

If you want to switch code depending on whether pthreads are available or not, you can use a feature identifier `gauche.sys.threads` with cond-expand form (see Section 4.12 [Feature conditional], page 75).

```scheme
(cond-expand
  [gauche.sys.threads
    ;; Code that uses thread API (gauche.threads is automatically
    ;; loaded at this moment).
  ]
  [else
    ;; Code that doesn’t use thread API
  ])
```

There are also feature identifiers `gauche.sys.pthreads` and `gauche.sys.wthreads` defined for pthreads and Windows threads platforms, respectively. In Scheme level, however, you hardly need to distinguish the underlying implementations. It is recommended to use `gauche.sys.threads` to switch the code according to thread availability.

To check if threads are available at runtime, instead of compile time, use the following procedure.

`gauche-thread-type` [Function]

{`gauche.threads`} Returns a symbol that indicates the supported thread type. It can be one of the following symbols.

- `none` Threads are not supported.
- `pthread` Threads are built on top of POSIX pthreads.
- `win32` Threads are built on top of Win32 threads.

(Note: On pthreads platforms, it should return `pthreads` instead of `pthread`; then the returned symbol would correspond to the value given to `--enable-threads` option at configuration time. It’s a historical overlook, stuck for the backward compatibility.)

Scheme-level thread API conforms SRFI-18, "Multithreading support", wrapped around Gauche’s object interface.

9.35.1 Thread programming tips

What’s Gauche threads for

Although the surface API of threads looks simple and portable, you need to know how the threads are implemented in order to utilize the feature’s potential. Some languages support threads as language’s built-in construct and encourage programmers to express the calculation in terms of threads. However, it should be noted that in many cases there are alternative ways
than threads to implement the desired algorithm, and you need to compare advantages and disadvantages of using threads depending on how the threads are realized in the underlying system.

In Gauche, the primary purpose of threads is to write programs that require preemptive scheduling, therefore are difficult to express in other ways. Preemptive threads may be required, for example, when you have to call a module that does blocking I/O which you can’t intercept, or may spend nondeterministic amount of calculation time that you want to interrupt.

For each Gauche’s thread, an individual VM is allocated and it is run by the dedicated POSIX thread. Thus the cost of context switch is the same as the native thread, but the creation of threads costs much higher than, say, lightweight threads built on top of call/cc. So Gauche’s preemptive threads are not designed for applications that want to create thousands of threads for fine-grained calculation.

The recommended usage is the technique so called "thread pool", that you create a set of threads and keep them around for long time and dispatch jobs to them as needed. Gauche provides a thread pool implementation in control.thread-pool module (see Section 12.12 [Thread pools], page 826).

Preemptive threads have other difficulties, and sometimes the alternatives may be a better fit than the native preemptive threads (e.g. see https://www-sop.inria.fr/mimosa/rp/FairThreads/html/FairThreads.html).

- If what you need is just a concurrent calculation, you might be able to use cooperative thread technique built on top of call/cc. Creating call/cc-based threads is much faster than creating native threads.
- If what you need is to deal with blocking I/O, and you have all your code at hand, it is sometimes easier to use good old select-based dispatching (See Section 9.30 [Simple dispatcher], page 509, for example).
- If what you need is to control the resource consumption in the subsystem, and the subsystem works fairly independently from the main system, you may be able to use Unix processes instead of threads. It may sound to go backward, but Unix process does provide higher "shield" between the subsystem and the main system (e.g. the main system can keep running even if subsystem segfaults).

Of course, these technique are not mutually exclusive with native threads. You can use dispatcher with "thread pool" technique, for example. Just keep it in your mind that the native threads are not only but one of the ways to realize those features.

### Uncaught errors in a thread body

When you run a single-thread program that raises an unexpected (unhandled) error, Gauche prints an error message and a stack trace by default. So sometimes it perplexes programmers when a thread doesn’t print anything when it dies because of an unhandled error.

What’s happening is this: An unhandled error in a thread body would cause the thread to terminate, and the error itself will propagate to the thread who’s expecting the result of the terminated thread. So, you get the error (wrapped by `<uncaught-exception>`) when you call thread-join! on a thread which is terminated because of an unhandled error. The behavior is defined in SRFI-18.

If you fire a thread for one-shot calculation, expecting to receive the result by thread-join!, then this is reasonable—you can handle the error situation in the “parent” thread. However, if you run a thread to loop indefinitely to process something and not expect to retrieve its result via thread-join!, this becomes a pitfall; the thread may die unexpectedly but you wouldn’t know it. (If such a thread is garbage-collected, a warning is printed. However you wouldn’t know when that happens so you can’t count on it.)
For such threads, you should always wrap the body of such thread with `guard`, and handles
the error explicitly. You can call `report-error` to display the default error message and a stack
trace.

\[
\text{(thread-start!}
  \quad \text{(make-thread } \left[\begin{array}{l}
\text{\texttt{]} (guard (e [else (report-error e) \#f])}
\quad \ldots \text{ thread body } \ldots
\end{array}\right]\text{))}
\]

See Section 9.35.5 [Thread exceptions], page 543, for the details of thread exception handling.

Note: As of 0.9.5, Gauche has a known bug that the tail call of error handling clauses of
`guard` doesn’t become a proper tail call. So, the following code, which should run safely in
Scheme, could eat up a stack:

\[
\text{(thread-start!}
  \quad \text{(make-thread } \left[\begin{array}{l}
\text{\texttt{]} (let loop ()}
\quad (guard (e [else (report-error e) (loop)])}
\quad \ldots \text{ thread body } \ldots
\end{array}\right]\text{))}
\]

For the time being, you can lift the call to loop outside of `guard` as workaround.

\[
\text{(thread-start!}
  \quad \text{(make-thread } \left[\begin{array}{l}
\text{\texttt{]} (let loop ()}
\quad (guard (e [else (report-error e)]))
\quad \ldots \text{ thread body } \ldots
\end{array}\right)]
\]

\[
\text{(loop))}
\]

### 9.35.2 Thread procedures

`<thread>`

A thread. Each thread has an associated thunk which is evaluated by a POSIX thread. When
thunk returns normally, the result is stored in the internal 'result' slot, and can be retrieved
by `thread-join!`. When thunk terminates abnormally, either by raising an exception or
terminated by `thread-terminate!`, the exception condition is stored in their internal 'result
exception' slot, and will be passed to the thread calling `thread-join!` on the terminated
thread.

Each thread has its own dynamic environment and dynamic handler stack. When a thread
is created, its dynamic environment is initialized by the creator’s dynamic environment. The
thread’s dynamic handler stack is initially empty.

A thread is in one of the following four states at a time. You can query the thread state by
the `thread-state` procedure.

- **new**: A thread hasn’t started yet. A thread returned from `make-thread` is in this
  state. Once a thread is started it will never be in this state again. At this point,
  no POSIX thread has been created; `thread-start!` creates a POSIX thread to
  run the Gauche thread.

- **Runnable**: When a thread is started by `thread-start!`, it becomes to this state. Note that
  a thread blocked by a system call is still in runnable state.

- **Stopped**: A thread becomes in this state when it is stopped by `thread-stop!`. A thread in
  this state can go back to runnable state by `thread-cont!`, resuming execution
  from the point when it is stopped.

- **terminated**: When the thread finished executing associated code, or is terminated by
  `thread-terminate!`, it becomes in this state. Once a thread is in this state,
  the state can no longer be changed.
Access to the resources shared by multiple threads must be protected explicitly by synchronization primitives. See Section 9.35.4 [Synchronization primitives], page 536.

Access to ports are serialized by Gauche. If multiple threads attempt to write to a port, their output may be interleaved but no output will be lost, and the state of the port is kept consistent. If multiple threads attempt to read from a port, a single read primitive (e.g. `read`, `read-char` or `read-line`) works atomically.

Signal handlers are shared by all threads, but each thread has its own signal mask. See Section 6.24.7.5 [Signals and threads], page 312, for details.

A thread object has the following external slots.

- **name**
  - [Instance Variable of `<thread>`]
  - A name can be associated to a thread. This is just for the convenience of the application. The primordial thread has the name "root".

- **specific**
  - [Instance Variable of `<thread>`]
  - A thread-local slot for use of the application.

- **current-thread**
  - [Function]
  - [SRFI-18], [SRFI-21]{gauche.threads} Returns the current thread.

- **thread? obj**
  - [Function]
  - [SRFI-18], [SRFI-21]{gauche.threads} Returns #t if obj is a thread, #f otherwise.

- **make-thread thunk ;optional name**
  - [Function]
  - [SRFI-18], [SRFI-21]{gauche.threads} Creates and returns a new thread to execute `thunk`. To run the thread, you need to call `thread-start!`. The result of `thunk` may be retrieved by calling `thread-join!`.

  You can provide the name of the thread by the optional argument `name`.

  The created thread inherits the signal mask of the calling thread (see Section 6.24.7.5 [Signals and threads], page 312), and has a copy of parameters of the calling thread at the time of creation (see Section 6.16.1 [Parameters], page 233).

  Other than those initial setups, there will be no relationship between the new thread and the calling thread; there's no parent-child relationship like Unix process. Any thread can call `thread-join!` on any other thread to receive the result. If nobody issues `thread-join!` and nobody holds a reference to the created thread, it will be garbage collected after the execution of the thread terminates.

  If a thread execution is terminated because of uncaught exception, and its result is never retrieved by `thread-join!`, a warning will be printed to the standard error port notifying “thread dies a lonely death”: It usually indicates some coding error. If you don’t collect the result of threads, you have to make sure that all the exceptions are captured and handled within `thunk`.

  Internally, this procedure just allocates and initializes a Scheme thread object; the POSIX thread is not created until `thread-start!` is called.

- **thread-state thread**
  - [Function]
  - {gauche.threads} Returns one of symbols `new`, `runnable`, `stopped` or `terminated`, indicating the state of `thread`.

- **thread-name thread**
  - [Function]
  - [SRFI-18], [SRFI-21]{gauche.threads} Returns the value of `name` slot of `thread`.

- **thread-specific thread**
  - [Function]
  - **thread-specific-set! thread value**
  - [SRFI-18], [SRFI-21]{gauche.threads} Gets/sets the value of the `thread`’s specific slot.
thread-start!  thread
[Function]  [SRFI-18], [SRFI-21]{gauche.threads} Starts the thread. An error is thrown if thread is not in “new” state. Returns thread.

thread-try-start!  thread
{gauche.threads} Starts and returns the thread if it is in “new” state. Otherwise, returns #f.

Note that a thread can become a “terminated” state even if it is never started, if another thread calls thread-terminate! on it. If that’s the possibility, this procedure comes handy, for thread-start! may raise an error if the thread is terminated before started.

thread-yield!
[Function]  [SRFI-18], [SRFI-21]{gauche.threads} Suspends the execution of the calling thread and yields CPU to other waiting runnable threads, if any.

thread-sleep!  timeout
[Function]  [SRFI-18], [SRFI-21]{gauche.threads} Suspends the calling thread for the period specified by timeout, which must be either a <time> object (see Section 6.24.9 [Time], page 316) that specifies absolute point of time, or a real number that specifies relative point of time from the time this procedure is called in number of seconds.

After the specified time passes, thread-sleep! returns with unspecified value.
If timeout points a past time, thread-sleep! returns immediately.

thread-stop!  thread :optional timeout timeout-val
{gauche.threads} Stops execution of the target thread temporarily. You can resume the execution of the thread by thread-cont!.

The stop request is handled synchronously; that is, Gauche VM only checks the request at the “safe” point of the VM and stops itself. It means if the thread is blocked by a system call, it won’t become stopped state until the system call returns.

By default, thread-stop! returns after the target thread stops. Since it may take indefinitely, you can give optional timeout argument to specify timeout. The timeout argument can be #f, which means no timeout, or a <time> object that specifies an absolute point of time, or a real number specifying the number of seconds to wait.

The return value of thread-stop! is thread if it could successfully stop the target, or timeout-val if timeout reached. When timeout-val is omitted, #f is assumed.

If the target thread has already been stopped by the caller thread, this procedure returns thread immediately.

When thread-stop! is timed out, the request remains effective even after thread-stop! returns. That is, the target thread may stop at some point in future. The caller thread is expected to call thread-stop! again to complete the stop operation.

An error is signaled if the target thread has already been stopped by another thread (including the “pending” stop request issued by other threads), or the target thread is in neither runnable nor stopped state.

thread-cont!  thread
{gauche.threads} Resumes execution of thread which has been stopped by thread-stop!. An error is raised if thread is not in stopped state, or it is stopped by another thread.

If the caller thread has already requested to stop the target thread but timed out, calling thread-cont! cancels the request.
thread-terminate! thread :key force

[Function]  
[SRFI-18*], [SRFI-21*]{gauche.threads} Terminates the specified thread thread. The thread is terminated and an instance of <terminated-thread-exception> is stored in the result exception field of thread.

If thread is the same as the calling thread, this procedure won’t return. Otherwise, this procedure returns unspecified value.

If thread is already terminated, this procedure does nothing.

By default, this procedure tries to terminate the thread in “safe” point, so that the thread can keep the process in consistent state. The locked mutexes, however, may remain locked; they become “abandoned” state and when other thread tries to touch the mutex it will raise <abandoned-mutex-exception>.

This strategy may, however, leave the system’s thread unterminated, if the thread is blocking in certain system calls, even in Scheme level the thread is marked terminated. The system thread will terminate itself as soon as it resumes and return the control to Gauche runtime.

If, for some reason, you need to guarantee to terminate the system thread, you can pass a true value to force keyword argument. If you do so, thread-terminate! will forcibly terminate the system thread after the usual graceful termination fails. It calls pthread_cancel or TerminateThread as the last resort, which can leave the process state inconsistent, so this mode should be used in an absolute emergency.

thread-join! thread :optional timeout timeout-val

[Function]  
[SRFI-18], [SRFI-21]{gauche.threads} Waits termination of thread, or until the timeout is reached if timeout is given.

Timeout must be either a <time> object (see Section 6.24.9 [Time], page 316) that specifies absolute point of time, or a real number that specifies relative point of time from the time this procedure is called in number of seconds, or #f that indicates no timeout (default).

If thread terminates normally, thread-join! returns a value which is stored in the result field of thread. If thread terminates abnormally, thread-join! raises an exception which is stored in the result exception field of thread. It can be either a <terminated-thread-exception> or <uncaught-exception>.

If the timeout is reached, thread-join! returns timeout-val if given, or raises <join-timeout-exception>.

See Section 9.35.5 [Thread exceptions], page 543, for the details of these exceptions.

9.35.3 Thread local storage

A thread local is an object that can hold per-thread value. It is defined in SRFI-226.

Note: Parameters used to be the way to keep per-thread value, but we’ve decided to split thread local storage from dynamic binding to eventually conform SRFI-226. Parameters should be used for dynamic binding, and thread locals should be used for per-thread value. If your existing code uses parameters for the per-thread value, update it to use thread locals. Parameters will eventually share the storage across threads.

make-thread-local init-value :optional inheritable? name

[Function]  
[SRFI-226]{gauche.threads} Returns a new thread local, with init-value as an initial value. The optional name argument specifies thread local’s name; it is only used for debugging purpose.

A thread local is a handle to thread specific storage that can contain single Scheme value. You can retrieve its value with tlref and set a new value with tlset!. Once a thread local is created, it can be used from all existing and future threads.
If #f is specified in `inheritable?` (default), when a thread local is `tlref`-ed first time in a thread, it returns `init-value`. If a true value is specified in `inheritable?`, on the other hand, the initial value of the thread local inherits the value of the creator thread at the moment the thread is created.

**thread-local? obj**

[Function]

SRFI-226}{gauche.threads} Returns #t if `obj` is a thread local, #f otherwise.

**tlref tl**

[Function]

SRFI-226}{gauche.threads} Returns the value of a thread local `tl`.

**tlset! tl val**

[Function]

SRFI-226}{gauche.threads} Modifies the value of a thread local `tl` with `val`.

As Gauche’s extension, you can also use `set!` with `tlref`:

\[
(set! (tlref tl) obj) \equiv (tlset! tl obj)
\]

### 9.35.4 Synchronization primitives

Mutexes and condition variables are the low-level synchronization devices. Defined in SRFI-18 and SRFI-21, they are portable across Scheme implementations that supports one of those srfis. (See Section 9.35.4.1 [Mutex], page 536, and see Section 9.35.4.2 [Condition variable], page 539, for the details.)

However, in most cases you want to use following higher-level synchronization utilities:

- **Atoms** An atom is a wrapper of arbitrary Scheme object, and allows synchronized access to it somewhat like Java’s `synchronized` blocks.

- **Semaphores** A traditional synchronization primitive to share fixed number of resources.

- **Latch** Holds thread(s) until a set of operations is completed by other thread(s).

- **Barrier** Wait until the preset number of thread reaches at the point.

- **MT-Queues** Thread-safe queues (`mtqueue`) are provided in `data.queue` module (see Section 12.19 [Queue], page 837), which works as a synchronized channel and suitable to implement producer-consumer pattern.

#### 9.35.4.1 Mutex

**<mutex>**

{gauche.threads} A primitive synchronization device. It can take one of four states: locked/owned, locked/not-owned, unlocked/abandoned and unlocked/not-abandoned. A mutex can be locked (by `mutex-lock!`) only if it is in unlocked state. An ‘owned’ mutex keeps a thread that owns it. Typically an owner thread is the one that locked the mutex, but you can make a thread other than the locking thread own a mutex. A mutex becomes unlocked either by `mutex-unlock!` or the owner thread terminates. In the former case, a mutex becomes unlocked/not-abandoned state. In the latter case, a mutex becomes unlocked/abandoned state.

A mutex has the following external slots.

**name**

[Instance Variable of `<mutex>`]

The name of the mutex.

**state**

[Instance Variable of `<mutex>`]

The state of the mutex. This is a read-only slot. See the description of `mutex-state` below.
specific

A slot an application can keep arbitrary data. For example, an application can implement a 'recursive' mutex using the specific field.

mutex? obj

[Function]

SRFI-18], [SRFI-21]{gauche.threads] Returns #t if obj is a mutex, #f otherwise.

make-mutex :optional name

[Function]

SRFI-18], [SRFI-21]{gauche.threads] Creates and returns a new mutex object. When created, the mutex is in unlocked/not-abandoned state. Optionally, you can give a name to the mutex.

mutex-name mutex

[Function]

SRFI-18], [SRFI-21]{gauche.threads} Returns the name of the mutex.

mutex-specific mutex

[Function]

mutex-specific-set! mutex value

[Function]

SRFI-18], [SRFI-21]{gauche.threads} Gets/sets the specific value of the mutex.

mutex-state mutex

[Function]

SRFI-18], [SRFI-21]{gauche.threads} Returns the state of mutex, which may be one of the followings:

- a thread The mutex is locked/owned, and the owner is the returned thread.
- symbol not-owned The mutex is locked/not-owned.
- symbol abandoned The mutex is unlocked/abandoned.
- symbol not-abandoned The mutex is unlocked/not-abandoned.

mutex-lock! mutex :optional timeout thread

[Function]

SRFI-18], [SRFI-21]{gauche.threads} Locks mutex. If mutex is in unlocked/not-abandoned state, this procedure changes its state to locked state exclusively. By default, mutex becomes locked/owned state, owned by the calling thread. You can give other owner thread as thread argument. If thread argument is given and #f, the mutex becomes locked/not-owned state.

If mutex is in unlocked/abandoned state, that is, some other thread has been terminated without unlocking it, this procedure signals 'abandoned mutex exception' (see Section 9.35.5 [Thread exceptions], page 543) after changing the state of mutex.

If mutex is in locked state and timeout is omitted or #f, this procedure blocks until mutex becomes unlocked. If timeout is specified, mutex-lock! returns when the specified time reaches in case it couldn’t obtain a lock. You can give timeout an absolute point of time (by <time> object, see Section 6.24.9 [Time], page 316), or a relative time (by a real number). Mutex-lock! returns #t if mutex is successfully locked, or #f if timeout reached.

Note that mutex itself doesn’t implements a ‘recursive lock’ feature; that is, if a thread that has locked mutex tries to lock mutex again, the thread blocks. It is not difficult, however, to implement a recursive lock semantics on top of this mutex. The following example is taken from SRFI-18 document:

```
(define (mutex-lock-recursively! mutex)
  (if (eq? (mutex-state mutex) (current-thread))
      (let ((n (mutex-specific mutex)))
        
```
(mutex-specific-set! mutex (+ n 1)))
(begin
(mutex-lock! mutex)
(mutex-specific-set! mutex 0)))

(define (mutex-unlock-recursively! mutex)
(let ((n (mutex-specific mutex)))
(if (= n 0)
(mutex-unlock! mutex)
(mutex-specific-set! mutex (- n 1))))

**mutex-unlock!**  \texttt{mutex :optional condition-variable timeout} \texttt{[Function]}

[SRFI-18], [SRFI-21] \{gauche.threads\} Unlocks \texttt{mutex}. The state of \texttt{mutex} becomes unlocked/not-abandoned. It is allowed to unlock a mutex that is not owned by the calling thread.

If optional \texttt{condition-variable} is given, \texttt{mutex-unlock!} serves the "condition variable wait" operation (e.g. \texttt{pthread_cond_wait} in POSIX threads). The current thread atomically wait on \texttt{condition-variable} and unlocks \texttt{mutex}. The thread will be unblocked when other thread signals on \texttt{condition-variable} (see \texttt{condition-variable-signal!} and \texttt{condition-variable-broadcast!} below), or \texttt{timeout} reaches if it is supplied. The \texttt{timeout} argument can be either a \texttt{<time>} object to represent an absolute time point (see Section 6.24.9 \{Time\}, page 316), a real number to represent a relative time in seconds, or \texttt{#f} which means never. The calling thread may be unblocked prematurely, so it should reacquire the lock of \texttt{mutex} and checks the condition, as in the following example (it is taken from SRFI-18 document):

(let loop ()
(mutex-lock! m)
(if (condition-is-true?)
(begin
(do-something-when-condition-is-true)
(mutex-unlock! m)))
(begin
(mutex-unlock! m cv)
(loop))))

The return value of \texttt{mutex-unlock!} is \texttt{#f} when it returns because of \texttt{timeout}, and \texttt{#t} otherwise.

**mutex-locker**  \texttt{mutex} \texttt{[Function]}

**mutex-unlocker**  \texttt{mutex} \texttt{[Function]}

\{gauche.threads\} Returns (lambda () (mutex-lock! mutex)) and (lambda () (mutex-unlock! mutex)), respectively. Each closure is created at most once per \texttt{mutex}, thus it is lighter than using literal lambda forms in a tight loop.

**with-locking-mutex**  \texttt{mutex thunk} \texttt{[Function]}

\{gauche.threads\} Calls \texttt{thunk} with locking a mutex \texttt{mutex}. This is defined as follows.

(define (with-locking-mutex mutex thunk)
(dynamic-wind
(mutex-locker mutex)
thunk
(mutex-unlocker mutex)))
9.35.4.2 Condition variable

A condition variable keeps a set of threads that are waiting for a certain condition to be true. When a thread modifies the state of the concerned condition, it can call \texttt{condition-variable-signal!} or \texttt{condition-variable-broadcast!}, which unblock one or more waiting threads so that they can check if the condition is satisfied.

A condition variable object has the following slots.

- **name**: The name of the condition variable.
- **specific**: A slot an application can keep arbitrary data.

Note that SRFI-18 doesn’t have a routine equivalent to pthreads’ \texttt{pthread_cond_wait}. If you want to wait on condition variable, you can pass a condition variable to \texttt{mutex-unlock!} as an optional argument (see above), then acquire mutex again by \texttt{mutex-lock!}. This design is for flexibility; see SRFI-18 document for the details.

This is the common usage of pthreads’ condition variable:

\begin{verbatim}
while (some_condition != TRUE) {
    pthread_cond_wait(condition_variable, mutex);
}
\end{verbatim}

And it can be translated to SRFI-18 as follows:

\begin{verbatim}
(let loop ()
    (unless some_condition
        (mutex-unlock! mutex condition-variable)
        (mutex-lock! mutex)
        (loop)))
\end{verbatim}

- \texttt{condition-variable? obj} \hspace{1cm} [Function]
  
  [SRFI-18], [SRFI-21] \texttt{gauche.threads} Returns \texttt{#t} if \texttt{obj} is a condition variable, \texttt{#f} otherwise.

- \texttt{make-condition-variable :optional name} \hspace{1cm} [Function]
  
  [SRFI-18], [SRFI-21] \texttt{gauche.threads} Returns a new condition variable. You can give its name by optional \texttt{name} argument.

- \texttt{condition-variable-name cv} \hspace{1cm} [Function]
  
  [SRFI-18], [SRFI-21] \texttt{gauche.threads} Returns the name of the condition variable.

- \texttt{condition-variable-specific cv} \hspace{1cm} [Function]
  
  [SRFI-18][SRFI-21] \texttt{gauche.threads} Gets/sets the specific value of the condition variable.

- \texttt{condition-variable-signal! cv} \hspace{1cm} [Function]
  
  [SRFI-18][SRFI-21] \texttt{gauche.threads} If there are threads waiting on \texttt{cv}, causes the scheduler to select one of them and to make it runnable.

- \texttt{condition-variable-broadcast! cv} \hspace{1cm} [Function]
  
  [SRFI-18][SRFI-21] \texttt{gauche.threads} Unblocks all the threads waiting on \texttt{cv}.
9.35.4.3 Atom

An atom is a convenient wrapper to make operations on a given set of objects thread-safe. Instead of defining thread-safe counterparts of every structure, you can easily wrap an existing data structures to make it thread-safe.

`atom val . . .` [Function]

{gauche.threads} Creates and returns an atom object with `val . . .` as the initial values.

`atom? obj` [Function]

{gauche.threads} Returns #t iff `obj` is an atom.

The following procedures can be used to atomically access and update the content of an atom. They commonly take optional `timeout` and `timeout-val` arguments, both are defaulted to #f. In some cases, the procedure takes more than one `timeout-val` arguments. With the default value #f as `timeout` argument, those procedures blocks until they acquire a lock.

The timeout arguments can be used to modify the behavior when the lock cannot be acquired in timely manner. `timeout` may be a `<time>` object (see Section 6.24.9 [Time], page 316) to specify an absolute point of time, or a real number to specify the relative time in seconds. If timeout is expired, those procedures give up acquiring the lock, and the value given to `timeout-val` is returned. In `atomic` and `atomic-update!`, you can make them return multiple timeout values, by giving more than one `timeout-val` arguments.

`atom-ref atom :optional index timeout timeout-val` [Function]

{gauche.threads} Returns `index`-th value of `atom`. See above for `timeout` and `timeout-val` arguments.

```
(define a (atom 'a 'b))

(atom-ref a 0) ⇒ a
(atom-ref a 1) ⇒ b
```

`atomic atom proc :optional timeout timeout-val timeout-val2 . . .` [Function]

{gauche.threads} Calls `proc` with the current values in `atom`, while locking `atom`. `proc` must take as many arguments as the number of values `atom` has.

The returned value(s) of `proc` is the result of `atomic`, unless timeout occurs. See above for `timeout` and `timeout-val` arguments.

For example, the `ref/count` procedure in the following example counts the number of times the hashtable is referenced in thread-safe way.

```
(define a (atom (make-hash-table 'eq?) (list 0)))

(define (ref/count a key)
  (atomic a
    (lambda (ht count-cell)
      (inc! (car count-cell))
      (hash-table-get h key))))
```

`atomic-update! atom proc :optional timeout timeout-val timeout-val2` [Function]

{gauche.threads} Calls `proc` with the current values in `atom` while locking `atom`, and updates the values in `atom` by the returned values from `proc`. `proc` must take as many arguments as the number of values `atom` has, and must return the at least the same number of values. If `proc` returns more than the number of values `atom` holds, the extra values are not used to update the `atom`, but is included in the return values of `atomic-update!`. 
The returned value(s) of `atomic-update!` is what `proc` returns, unless timeout occurs. See above for `timeout` and `timeout-val` arguments.

The following example shows a thread-safe counter.

```
(define a (atom 0))

(atomic-update! a (cut + 1 <>))
```

Note: The term `atom` in historical Lisps meant an object that is not a cons cell (pair). Back then cons cells were the only aggregate datatype and there were few other datatypes (numbers and symbols), so having a complementary term to cells made sense.

Although it still appears in introductory Lisp tutorials, modern Lisps, including Scheme, has so many datatypes and it makes little sense to have a specific term for non-pair types.

Clojure adopted the term `atom` for thread-safe (atomic) primitive data, and we followed it.

Note: The constructor of atom is not `make-atom` but `atom`, following the convention of `list/make-list`, `vector/make-vector`, and `string/make-string`; that is, the name without `make-` takes its elements as variable number of arguments.

### 9.35.4.4 Semaphore

`<semaphore>` [Builtin Class]

{gauche.threads} A semaphore manages “tokens” for a fixed number of resources. A thread that wants to use one of the resources requests a token by `semaphore-acquire!`, and returns the token to the pool by `semaphore-release!`. When the thread requests a token but none is available, it waits until a token becomes available by some other thread returning one.

`make-semaphore :optional init-value name` [Function]

{gauche.threads} Creates and returns a new semaphore, with `init-value` tokens available initially. If `init-value` is omitted, 0 is assumed. Another optional argument `name` is an arbitrary Scheme object and only used for a debugging purpose only; it is displayed when a semaphore is printed.

`semaphore? obj` [Function]

{gauche.threads} Returns `#t` iff `obj` is a semaphore.

`semaphore-acquire! sem :optional timeout timeout-val` [Function]

{gauche.threads} Obtain a token from a semaphore `sem`. If a token is available, this returns `#t` immediately, decrementing the available token count of `sem` atomically. If no token is available, this waits until a token becomes available, or `timeout` reaches if it is given. If `timeout` occurs, it returns `timeout-val`, defaulted to `#f`.

The value of `timeout` is the same as `mutex`; it can be `#f` (no timeout, default), a `<time>` object to specify an absolute point of time, or a real number to specify the minimum number of seconds to wait.

`semaphore-release! sem :optional count` [Function]

{gauche.threads} Return a token to a semaphore `sem`, by incrementing its token count, and running one of the threads waiting with `semaphore-acquire!` atomically.

The `count` argument specifies number of tokens to return, defaulted to 1.

Actually, the “token” model is to make it easier to understand the idea of semaphore, but internally it’s just a counter, so you can call `semaphore-release!` even you haven’t called `semaphore-acquire!`. 
Chapter 9: Library modules - Gauche extensions

9.35.4.5 Latch

<latch>  {gauche.threads} A latch is a synchronization device with a counter. Any number of threads can wait until the counter becomes zero. Once the counter reaches zero, all threads go. (As opposed to semaphores, whose internal counter affects number of waiting threads as well.)

One simple usage is to start with initial count 1; it is sometimes called a “gate”. You can let threads wait before the gate, and decrement the counter, then boom! All threads proceed.

make-latch initial-count :optional name  {gauche.threads} Creates and returns a new latch with the counter value initial-count, which must be an exact positive integer.

The optional name argument can be any Scheme object, and used only for debugging. It is displayed when the latch object is printed.

latch? obj  {gauche.threads} Returns #t iff obj is a latch.

latch-dec! latch :optional n  {gauche.threads} Decrement the counter of latch by n. If n is omitted, 1 is assumed.

If the value of the latch becomes 0 or less, threads waiting on the latch are woken up.

Returns the updated counter value.

N must be an exact integer. Zero or negative value is allowed.

latch-clear! latch  {gauche.threads} If latch’s count was non-zero, make it zero and wakes up the threads waiting on it. If latch’s count has already been zero, do nothing.

Returns the previous count.

latch-await latch :optional timeout timeout-val  {gauche.threads} If the counter of latch is zero or negative, returns #t immediately. Otherwise, the calling thread is blocked until the counter value becomes zero or below, or timeout reaches. If the thread started by the counter value, #t is returned. If the thread started by timeout, timeout-val is returned, whose default is #f.

The timeout argument must be either #f (no timeout, default), a <time> object (absolute point of time), or a real number indicates number of seconds from the time this procedure is called.

9.35.4.6 Barrier

<barrier>  {gauche.threads} A barrier is a synchronization primitive that multiple threads wait by barrier-await until a specified number of threads reach the point. Once the specified number of threads get to the barrier, all threads go, and the barrier automatically returns to the initial state; the next thread that comes to the barrier waits again.

A barrier can have an optional action, which is run by the last thread that reached to the barrier before everyone goes.

A barrier can be in a “broken” state when any one of the threads timeout, or the action raises an uncaptured exception. Once the barrier is broken, all threads waiting on it is released as if timeout has occured. The broken barrier remains broken (so any thread that reaches it just passes through) until it is reset by barrier-reset!.
make-barrier threshold :optional action name
{gauche.threads} Creates and returns a new barrier. The threshold argument must be an exact nonnegative integer to specify the number of threads to let them go—threads calling barrier-await waits until the number of threads waiting reaches threshold.

The optional action argument must be either #f (no action) or a thunk. If it is a thunk, it is run by the last thread that calls barrier-await, before all the thread will go.

The optional name argument can be any Scheme object, and only used to print the barrier instance, to help debugging.

barrier? obj
{gauche.threads} Returns #t iff obj is a barrier.

barrier-await barrier :optional timeout timeout-val
{gauche.threads} Waits on barrier until the number of waiting threads reaches the threshold of the barrier, or the barrier is broken.

A barrier can be broken if any of waiting threads hit the timeout, or an uncaptured exception is raised from the action thunk of the barrier.

The timeout and timeout-val are the same as other synchronization primitives; timeout may be #f (no timeout, default), a <time> object to specify an absolute time point, or a real number indicating relative time in number of seconds; timeout-val is the value returned when barrier-await returns prematurely—either timeout, or other barrier breakage conditions.

The default of timeout-val is #f.

It returns an integer if the thread is released by reaching the threshold, or timeout-val if the barrier is broken. The integer indicates the number of threads required to reach the threshold at the time barrier-await is called; that is, if you’re the first to reach the barrier it is threshold - 1, and if you’re the last it is 0.

If the barrier is already broken when you call barrier-await, it returns immediately with timeout-val.

If the threads are released normally, the barrier turns back to the initial state so you can wait on it again. If the threads are released because the barrier is broken, it remains in broken state until reset by barrier-reset!.

barrier-broken? barrier
{gauche.threads} Returns #t iff barrier is in broken state.

barrier-reset! barrier
{gauche.threads} Resets the barrier to the initial state. If the barrire is broken, the broken state is cleared. If any thread is waiting on the barrier, it is released as if the number of threads reaches the threshold.

9.35.5 Thread exceptions

Some types of exceptions may be thrown from thread-related procedures. These exceptions can be handled by Gauche’s exception mechanism (see Section 6.19 [Exceptions], page 245).

These are originally defined in SRFI-18. SRFI-226 follows them, but uses slightly different names for the consistency with modern Scheme. We support both.

<thread-exception>
{SRFI-226}{gauche.threads} A base class of thread-related exceptions. Inherits <exception> class. It has one slot.

thread
A thread that threw this exception.
The name &thread is defined by SRFI-226. Note that SRFI-226 does not define thread slot; it’s Gauche’s extension.

<join-timeout-exception> [Builtin Class]  
&thread-timeout [Condition Type]  
|SRFI-226|{gauche.threads} An exception thrown by thread-join! when a timeout reaches before the waited thread returns. Inherits <thread-exception>.  

The name &thread-timeout is defined by SRFI-226.

<abandoned-mutex-exception> [Builtin Class]  
&thread-abandoned-mutex [Condition Type]  
|SRFI-226|{gauche.threads} An exception thrown by mutex-lock! when a mutex to be locked is in unlocked/abandoned state. Inherits <thread-exception>. It has one additional slot.  

mutex [Instance Variable of <abandoned-mutex-exception>]  
A mutex that caused this exception.

The name &thread-abandoned-mutex is defined by SRFI-226.

<terminated-thread-exception> [Builtin Class]  
&thread-already-terminated [Condition Type]  
|SRFI-226|{gauche.threads} An exception thrown by thread-join! when the waited thread is terminated abnormally (by thread-terminate!). Inherits <thread-exception>. It has one additional slot.  

terminator [Instance Variable of <terminated-thread-exception>]  
A thread that terminated the thread that causes this exception.

The name &thread-already-terminated is defined by SRFI-226. SRFI-226 doesn’t define terminator slot, though.

<uncaught-exception> [Builtin Class]  
&uncaught-exception [Condition Type]  
|SRFI-226|{gauche.threads} An exception thrown by thread-join! when the waited thread is terminated by an uncaught exception. Inherits <thread-exception>. It has one additional slot.  

reason [Instance Variable of <uncaught-exception>]  
An exception that caused the termination of the thread.

The name &uncaught-exception is defined by SRFI-226.

<concurrent-modification-violation> [Builtin Class]  
&concurrent-modification [Condition Type]  
|SRFI-226| This is a condition thrown when the runtime detects non-thread-safe data is being modified by multiple threads concurrently. SRFI-226 defines this, but it allows implementations not to detect this situation if it’s costly. Currently, no Gauche APIs detect this condition, though it may do so in future. For now, this is provided for the compatibility with SRFI-226.

make-thread-condition [Function]  
|SRFI-226|{gauche.threads} Creates a new <thread-exception> condition. This is for SRFI-226 compatibility. Usually you don’t need to use this directly, for the actual condition is of a subclass of <thread-exception>. The thread slot is initialized with the calling thread.
thread-condition? obj
[Function]
[SRFI-226]{gauche.threads} Returns \#t if obj is a \textless thread-exception\textgreater{} (\&thread) condition, \#f otherwise.

make-thread-timeout-condition
[Function]
[SRFI-226]{gauche.threads} Creates a new \textless join-timeout-exception\textgreater{} (\&thread-timeout) condition.

join-timeout-exception? obj
thread-timeout-condition? obj
[Function]
[Function]
[SRFI-18], [SRFI-21], [SRFI-226]{gauche.threads} Returns \#t if obj is a \textless join-timeout-exception\textgreater{} (\&thread-timeout) condition, \#f otherwise. The name join-timeout-exception? is defined by SRFI-18/21, and thread-timeout-condition? is by SRFI-226. They are the same.

make-thread-abandoned-mutex
[Function]
[SRFI-226]{gauche.threads} Creates a new \textless abandoned-mutex-exception\textgreater{} (\&thread-abandoned-mutex) condition.

abandoned-mutex-exception? obj
thread-abandoned-mutex-condition? obj
[Function]
[Function]
[SRFI-18], [SRFI-21], [SRFI-226]{gauche.threads} Returns \#t if obj is a \textless abandoned-mutex-exception\textgreater{} (), \#f otherwise. The name abandoned-mutex-exception? is defined by SRFI-18/21, and thread-abandoned-mutex-condition? is by SRFI-226. They are the same. Creates a new \textless terminated-thread-exception\textgreater{} (\&thread-already-terminated) condition.

terminated-thread-exception? obj
thread-already-terminated-condition? obj
[Function]
[Function]
[SRFI-18], [SRFI-21], [SRFI-226]{gauche.threads} Returns \#t if obj is an instance of \textless terminated-thread-exception\textgreater{}, \#f otherwise. The name terminated-thread-exception? is defined by SRFI-18/21, and thread-already-terminated-condition? is by SRFI-226. They are the same. Creates a new \textless uncaught-exception\textgreater{} (\&uncaught-exception) condition. The argument is set to the reason slot of the condition.

uncaught-exception? obj
uncaught-exception-condition? obj
[Function]
[Function]
[SRFI-18], [SRFI-21], [SRFI-226]{gauche.threads} Returns \#t if obj is an instance of \textless uncaught-exception\textgreater{}, \#f otherwise. The name uncaught-exception? is defined by SRFI-18/21, and uncaught-exception-condition? is by SRFI-226. They are the same. Returns the value of reason slot of \textless uncaught-exception\textgreater{} object.

The name uncaught-exception-reason is defined in SRFI-18/21, and uncaught-exception-condition-reason is defined in SRFI-226. Both works the same.

make-concurrent-modification-violation
[Function]
[SRFI-226]{gauche.threads} Creates a new \textless concurrent-modification-violation\textgreater{} (\&concurrent-modification) condition.

concurrent-modification-violation? obj
[Function]
[SRFI-226]{gauche.threads} Returns \#t if obj is a \textless concurrent-modification-violation\textgreater{} (\&concurrent-modification) condition, \#f otherwise.
9.36 gauche.time - Measure timings

**gauche.time**

Provides three ways to measure execution time of Scheme code. A macro `time`, which is convenient for interactive use, a set of procedures for benchmarking, and `<time-counter>` objects which are useful to be embedded in the program.

**Interactive measurement of execution time**

Note: The `time` macro is pre-defined to autoload `gauche.time` for the convenience; you don’t need to say `(use gauche.time)` to use the `time` macro.

```scheme
(time expr expr2 ...)
```

Evaluates `expr expr2 ...` sequentially, as `begin`, and returns the result(s) of the last expression. Before returning the value(s), the macro reports the elapsed (real) time and CPU times in the user space and the kernel space to the current error port, much like the bourne shell’s `time` command.

The current version uses `sys-gettimeofday` (see Section 6.24.9 [Time], page 316) to calculate the elapsed time, and `sys-times` (see Section 6.24.8 [System inquiry], page 313) to calculate user and system CPU times. So the resolution of these numbers depends on these underlying system calls. Usually the CPU time has 10ms resolution, while the elapsed time might have higher resolution. On the systems that doesn’t have gettimeofday(2) support, however, the elapsed time resolution can be as bad as a second.

```scheme
gosh> (time (length (sort (call-with-input-file "/usr/share/dict/words" port->string-list))))
```

```
; (time (length (sort (call-with-input-file "/usr/share/dict/words" port- ... 
; real 0.357
; user 0.350
; sys 0.000
45427
```

**Benchmarking**

It is not unusual that the routine you want to measure takes only a fraction of second, so you have to run it many times for better measurement. It is also common that you want to compare results of measurement of two or more implementation strategies. Here are useful procedures to do so.

The name and behavior of those benchmarking routines are inspired by Perl’s Benchmark module.

```scheme
(time-this how thunk)
```

Calls `thunk` many times and measure its execution time. The argument `how` can be one of the following forms.

- **integer**
  It calls `thunk` as many times as the given number.

- **(metric real)**
  It calls `thunk` as many times as the total time exceeds the given number of seconds. `Metric` is either one of symbols `cpu`, `user`, `sys`, or `real`, to specify how the time is measured.

It also runs an empty loop as the same times and subtract the time took for the empty loop from the measured time, to get more accurate result.

The result is returned in a `<time-result>` record, described below. Here are some examples:

```scheme
;; Run the thunk 1,000,000 times
(time-this 1000000 (lambda () (expt 100 30)))
⇒ #<time-result 1000000 times/ 1.030 real/ 1.040 user/ 0.000 sys>
```
;;; Run the thunk at least 5.0 cpu seconds
(time-this '(cpu 5.0) (lambda () (expt 100 30)))
⇒ #<time-result 4903854 times/ 5.090 real/ 5.050 user/ 0.010 sys>

<time-result>                  [Record]
{gauche.time} A record to hold the benchmark result. Following slots are defined.

  count                  [Instance Variable of <time-result>]
    The number of times the thunk was run. This slot is also accessed by a procedure
    time-result-count.

  real                   [Instance Variable of <time-result>]
    The total real (elapsed) time running the thunk took. This slot is also accessed by a
    procedure time-result-real.

  user                   [Instance Variable of <time-result>]
    The total user cpu time running the thunk took. This slot is also accessed by a procedure
    time-result-user.

  sys                    [Instance Variable of <time-result>]
    The total system cpu time running the thunk took. This slot is also accessed by a procedure
    time-result-sys.

make-time-result  count real user sys          [Function]
{gauche.time} The constructor of <time-result> records.

time-result?  obj                    [Function]
{gauche.time} The predicate of <time-result> records.

time-result+  t1 t2 :key (with-count #f)    [Function]
{gauche.time} Add or subtract two <time-result> records and returns a new record.
If with-count is false, only the real, user and sys slots are added or subtracted, and the result’s
count slot is set to the same as t1’s count slot. It is supposed to be used to calculate on
measurement from different chunk of code.
If with-count is true, then the values of count slot is also added or subtracted. It is supposed
to calculate on multiple benchmark results of the same code.

  time-these  how alist          [Function]
  time-these/report  how alist :key metric [Function]
{gauche.time} These procedures benchmarks multiple chunks of code to compare.
The alist argument must be the form of ((key . thunk) ...), where key is a symbol and
thunk is a procedure taking no arguments.
The how argument is the same as time-this; that is, either an integer for number of iterations,
or a list (cpu x) to indicate x seconds of cpu time.

  time-these runs benchmarks for each thunk in alist using time-this, and returns the result
in a list of the form (how (key1 . result1) (key2 . result2) ...), where each result is a
<time-result> object.

  time-these/report outputs the benchmark results and comparison matrix in human readable
way to the current output port.

gosh> (time-these/report '(cpu 3.0)
   '((real1 . ,(cut expt 100 20))
     (real2 . ,(cut %expt 100 20))
     (imag . ,(cut expt +100i 20)))))
Benchmark: ran real1, real2, imag, each for at least 3.0 cpu seconds.
real1: 3.312 real, 3.320 cpu (3.320 user + 0.000 sys)@ 1694277.11/s n=5625000
real2: 2.996 real, 3.010 cpu (3.010 user + 0.000 sys)@35595635.55/s n=107142860
imag: 3.213 real, 3.190 cpu (3.190 user + 0.000 sys)@ 862068.97/s n=2750000

Rate   real1   real2   imag
real1  1694277/s -- 0.048  1.965
real2  35595635/s 21.009 -- 41.291
imag   862069/s  0.509  0.024 --

The first part of the report shows, for each thunk, the real (elapsed) time, the cpu time used
(and its breakdown of user and system time), the rate of iteration per second, and the total
number of iterations.

The second part compares the speed between each pair of the benchmarks. For example, its
first row tells that the benchmark real1 is 0.048 times faster than real2 and 1.965 times
faster than imag.

The metric argument specifies which time metric should be used for rate calculation and
comparison. It must be either #f or one of the symbols cpu, user, sys, or real. If it is
#f (default), the same metric as how argument is used (if how is an integer count, cpu is
assumed).

report-time-results result :optional metric

{gauche.time} This is a utility procedure to create a report from the result of time-these.
Actually, time-these/report is just a combination of time-these and this procedure:
The meaning of the optional metric argument is the same as the metric keyword argument
of time-these/report.

(define (time-these/report how samples :key metric)
  (report-time-results (time-these how samples) metric))

Finer measurement
<time-counter>
{gauche.time} An abstract class of time counters. Time counter is a kind of timer whose
value is incremented as the time passes. The counting can be started and stopped any number
of times. The value of the counter can be read when the timer is stopping. You can have
multiple time counters. It is useful, for example, to measure the time in two parts inside a
loop independently.

The concrete subclass determines which time it is counting. You have to instantiate one of
those subclasses described below to use the time counter.

<real-time-counter> {gauche.time} Classes for time counters that count real (elapsed) time, user-space CPU
time, kernel-space CPU time, and total CPU time (user + system), respectively.

time-counter-start! (counter <time-counter>) {gauche.time} Starts and stops the counter. The time during the counter is running is
accumulated to the counter value when the counter is stopped.

Start/stop pairs can be nested, but only the outermost pair takes the effect. That is, if you
call time-counter-start! on the counter that is already started, it doesn’t have any effect
except that to stop such a counter you have to call time-counter-stop! one more time.
It is useful when you want to measure the time spent in the larger block that may already contain timer start/stop pairs.

Calling `time-counter-stop!` on the already stopped counter has no effect.

```scheme
(time-counter-reset! (counter <time-counter>)
{gauche.time} Resets the value of `counter`. If `counter` is already running, it is forced to stop before being reset.

(time-counter-value (counter <time-counter>)
{gauche.time} Returns the current value of the counter as the number of seconds, in a real number. The resolution depends on the source of the counter.

(with-time-counter counter expr ...)
{gauche.time} A convenience macro to run the `counter` while `expr ...` are evaluated. Returns the result(s) of the last expression. It is defined as follows.

```scheme
(define-syntax with-time-counter
  (syntax-rules ()
    ((_ counter . exprs)
      (dynamic-wind
       (lambda () (time-counter-start! counter))
       (lambda () . exprs)
       (lambda () (time-counter-stop! counter))))
))
```

The following example measures approximate times spend in process-A and process-B inside a loop.

```scheme
(let ((ta (make <real-time-counter>))
  (tb (make <real-time-counter>))
  (do ((i 100000))
    ((time-counter-value ta)
      (time-counter-value tb))
    (format #t "Time spent in process-A: " (time-counter-value ta))
    (format #t "Time spent in process-B: " (time-counter-value tb))
))
```

9.37 `gauche.unicode` - Unicode utilities

`gauche.unicode` [Module]

This module provides various operations on a sequence of Unicode codepoints.

Gauche can be compiled with a native encoding other than Unicode, and the full Unicode-compatible behavior on characters and strings may not be available on such systems. So we provide most operations in two flavors: Operations on characters and strings, or operations on codepoints represented as a sequence of integers.

If Gauche is compiled with its native encoding being `none`, `euc-jp` or `sjis`, character-and-string operations are likely to be partial functions of the operations defined in Unicode standard. That is, if the operation can yield a character that are not supported in the native encoding, it may be remapped to an alternative character. Each manual entry explains the detailed behavior.

The codepoint operations are independent from Gauche’s native encoding and supports full spec as defined in Unicode standard. If Gauche is compiled with the `utf-8` native encoding,
the operations are essentially the same as character-and-string flavors when you convert
codepoints and characters by char->integer and integer->char. The codepoint operations
are handy when you need to support the algorithms described in Unicode standard fully, no
matter what the running Gauche’s native encoding is.

9.37.1 Unicode transfer encodings
The procedures in this group operate on codepoints represented as integers. In the following
descriptions, an octet refers to an integer between 0 to 255, inclusive.

They take optional strictness argument. It specifies what to do when the procedure en-
counters a datum outside of the defined domain. Its value can be either one of the following
symbols:

strict Raises an error when the procedure encounters such input. This is the default
behavior.

permissive Whenever possible, treat the date as if it is a valid value. For example, codepoint
value beyond #x10ffff is invalid in Unicode standard, but it may be useful for some
other purpose that just want to use UTF-8 as an encoding scheme of binary data.

replace If the procedure sees invalid input, replaces it with a unicode replacement character
U+FFFD and proceed.

ignore Whenever possible, treat the invalid input as if they do not exist.

The procedure may still raise an error in permissive, replace or ignore strictness mode,
if there can’t be a sensible way to handle the input data.

ucs4->utf8 codepoint :optional strictness
{gauche.unicode} Takes an integer codepoint and returns a list of octets that encodes the
input in UTF-8.

(ucs4->utf8 #x3bb) ⇒ (206 187)
(ucs4->utf8 #x3042) ⇒ (227 129 130)

If strictness is strict (default), input codepoint between #xd800 to #xdfff, and beyond
#x110000, are rejected. If strictness is replace, such input yields a utf8 sequence #xef,
#xbf, #xbd, which encodes U+FFFD. If strictness is permissive, it accepts input between 0
and #x7fffffff, inclusive; it may produce 5 or 6 octets if the input is large (as the original
UTF-8 definition). If strictness is ignore, it returns an empty list for invalid codepoints.

utf8-length octet :optional strictness
{gauche.unicode} Takes octet as the first octet of UTF-8 sequence, and returns the number
of total octets required to decode the codepoint.

If octet is not an exact integer between 0 and 255 (inclusive), an error is thrown, regardless
of strictness argument.

If strictness is strict (default), this procedure returns either 1, 2, 3 or 4. An error is thrown
if octet cannot be a leading octet of a proper UTF-8 encoded Unicode codepoint.

If strictness is permissive or replace, this procedure may return an integer between 0 and
6, inclusive. If the input is from #xf8 to #xfd, inclusive, this returns 5 or 6, according to the
original utf-8 spec (these values corresponds to the codepoint range #x110000 to #x7fffffff).
If the input is in the range between #x80 and #xbf, inclusive, or #xe or #xff, this procedure
returns 1–it’s up to the application how to treat these illegal octets.

If strictness is ignore, this procedure returns 0 when it would raise an error if strictness is
strict. Other than that, it works the same as the default case.
utf8->ucs4  octet-list :optional strictness  
{gauche.unicode} Takes a list of octets, and decodes it as a utf-8 sequence. Returns two values: The decoded ucs4 codepoint, and the rest of the input list.

If it finds a value other than exact integer between 0 and 255 in the input list, an error is thrown regardless of the value of strictness.

An invalid utf8 sequence causes an error if strictness is strict, or skipped if it is ignore. If strictness is replace, such utf8 sequence yields U+FFFD. If strictness is permissive, the procedure accepts the original utf-8 sequence which can produce surrogated pair range (between #xd800 and #dfff) and the range between #x110000 to #x7fffffff. The invalid octet sequence is still an error with permissive mode.

utf8->string  u8vector :optional start end  
[R7RS base] {gauche.unicode} Converts a sequence of utf8 octets in u8vector to a string. Optional start and/or end argument(s) will limit the range of the input.

If Gauche’s native encoding is utf8, this procedure first tries u8vector->string (see Section 6.13.2 [Uniform vectors], page 198). If the input utf8 sequence is valid, this is the fastest way. If the input contains invalid utf8 sequence, the procedure falls back to construct a string by one character at at time, replacing invalid sequence with Unicode replacement character U+FFFD. Hence it always returns a complete string. To know if the input contains invalid utf8 sequence, you can use u8vector->string directly.

If Gauche’s native encoding is other than utf8, there’s no U+FFFD so invalid utf8 sequence throws an error.

string->utf8  string :optional start end  
[R7RS base] {gauche.unicode} Converts a string to a u8vector of utf8 octets. Optional start and/or end argument(s) will limit the range of the input.

If Gauche’s native encoding is utf8, this procedure just calls string->u8vector (see Section 6.13.2 [Uniform vectors], page 198). Otherwise, it first converts the input string to utf-8, then string->u8vector is called.

ucs4->utf16  codepoint :optional strictness  
{gauche.unicode} Takes an integer codepoint and returns a list of integers that encodes the input in UTF-16.

If strictness is strict (default), the input must be either between 0 and #xd7ff or between #xe000 and #x10ffff. An error is thrown otherwise. The ‘hole’ is the codepoint reserved for surrogates, and there’s no valid mapping from them to utf-16 is defined.

If strictness is replace, such input is replaced with #xfffd, which encodes Unicode replacement character.

If strictness is permissive, it accepts high surrogates and low surrogates, in which case the result is single element list of input. An error is still thrown for negative input and input greater than or equal to #x110000.

If strictness is ignore, an empty list is returned for an invalid codepoint (including surrogates).

Note: We can encode values larger than #x10ffff in utf-8 in the permissive mode, but not in utf-16.

utf16-length  code :optional strictness  
{gauche.unicode} Code must be an exact integer between 0 and 65535, inclusive. Returns 1 if code is BMP character codepoint, or 2 if code is a high surrogate.

If strictness is strict (default), an error is signalled if code is a low surrogate, or it is out of range. If strictness is permissive or replace, 1 is returned for low surrogates, but an
error is signalled for out of range arguments. If strictness is ignore, 0 is returned for low surrogates and out of range arguments.

**utf16->ucs4** code-list :optional strictness

{gauche.unicode} Takes a list of exact integers and decodes it as a utf-16 sequence. Returns two values: The decoded ucs4 codepoint, and the rest of input list.

If strictness is strict (default), an invalid utf-16 sequence and out-of-range integer raise an error. If strictness is permissive, an out-of-range integer causes an error, but a lone surrogate is allowed and returned as is. If strictness is replace, a lone surrogate is replaced with \texttt{U+FFFD}. If strictness is ignore, lone surrogates and out-of-range integers are just ignored.

**utf16->string** u8vector :optional endian ignore-bom? start end

{guache.unicode} [R7RS scheme.bytevector] Convert utf16 and utf32 sequence stored in u8vector to a string, respectively. For utf16->string, if the input contains invalid utf16 sequence (unpaired surrogate), it is replaced with Unicode replacement character \texttt{U+FFFD}. If the number of input octet is not the multiple of unit (2 octets for utf16, and 4 octets for utf32), an error is thrown.

The optional endian and ignore-bom? arguments determines whether the input is in UTF16BE/UTF32BE or UTF16LE/UTF32LE. If ignore-bom? is \#f or omitted, the first two octets of input is examined; if it’s BOM, it determines the endianness regardless of endian argument, and the BOM won’t be included in the output. If the input does not begin with BOM, or ignore-bom? is true, then the endianness is determined by endian argument: It can be big-endian or big for UTF16BE/UTF32BE, and little-endian or little or arm-big-endian for UTF16LE/UTF32LE (see Section 6.3.7 [Endianness], page 138, for the details of endianness).

Note that if ignore-bom? is given and true, the initial BOM is interpreted as a codepoint \texttt{U+FEFF}. If endian is \#f or omitted, UTF16BE/UTF32BE is assumed (it is defined so in R7RS scheme.bytevector).

In R7RS scheme.bytevector, ignore-bom? argument is called endianness-mandatory. The behavior is the same.

Optional argument start and end trims the input octet sequence before other processing (including BOM detection). These arguments are Gauche’s extension, and not the part of R7RS scheme.bytevector.

**string->utf16** str :optional endian add-bom? start end

{gauche.unicode} [R7RS scheme.bytevector] Encode a string str to utf-16 and utf-32 sequences stored in a u8vector, respectively.

The optional endian argument specifies whether the encoding is UTF16BE/UTF32BE or UTF16LE/UTF32LE. If it is a symbol big-endian or big, the encoding is UTF16BE/UTF32BE. If it is a symbol little-endian, little or arm-little-endian, the encoding is UTF16LE/UTF32LE. See Section 6.3.7 [Endianness], page 138, for the details of endianness. If it is omitted or \#f, UTF16BE/UTF32BE is assumed.

The second optional argument add-bom? specifies, if true value is given, the output contains BOM. When omitted BOM won’t be added.

The start and end arguments limits the range of str to be converted.

R7RS scheme.bytevector only defines endian optional argument. The rest is Gauche’s extension.
### 9.37.2 Unicode text segmentation

These procedures implements grapheme-cluster and word breaking algorithms defined in UAX #29: Unicode Text Segmentation.

#### string->words

%Function%  
\texttt{string->words \textit{string}}

\{\texttt{gauche.unicode}\} From given string or codepoint sequence (a \texttt{sequence} object containing codepoints), returns a list of words. Each word is represented as a string, or a sequence of the same type as input, respectively.

\begin{verbatim}
(string->words "That's it."
⇒ ("That's" " " "it" ".")
\end{verbatim}

\begin{verbatim}
(codepoints->words '(84 104 97 116 39 115 32 105 116 46)
⇒ ((84 104 97 116 39 115) (32) (105 116) (46))
\end{verbatim}

\begin{verbatim}
(codepoints->words '#(84 104 97 116 39 115 32 105 116 46)
⇒ (#(84 104 97 116 39 115) #(32) #(105 116) #(46))
\end{verbatim}

In the second and third example, the input is a sequence of codepoints of characters in "That’s it."

#### string->grapheme-clusters

%Function%  
\texttt{string->grapheme-clusters \textit{string}}

\{\texttt{gauche.unicode}\} From given string or codepoint sequence (a \texttt{sequence} object containing codepoints), returns a list of grapheme clusters. Each cluster is represented as a string, or a sequence of the same type as input, respectively.

\begin{verbatim}
(string->grapheme-clusters "That's it."
⇒ ("That's" " " "it" ".")
\end{verbatim}

The following procedures are low-level building blocks to build the above \texttt{string->words} etc.

#### make-word-breaker

%Function%  
\texttt{make-word-breaker \textit{generator}}

\{\texttt{gauche.unicode}\} From given generator, which is a generator of characters or codepoints, returns a generator that returns two values: The first value is the character or codepoint generated from the original generator, and the second value is a boolean flag, which is \texttt{#t} if a word or a grapheme cluster breaks before the character/codepoint, and \texttt{#f} otherwise.

Suppose a generator \texttt{g} returns characters in a string \texttt{That’s it.}, one at a time. Then the created generator will work as follows:

\begin{verbatim}
(define brk (make-word-breaker g))
\end{verbatim}

\begin{verbatim}
(brk) ⇒ #\T and #t
(brk) ⇒ #\h and #f
(brk) ⇒ #\a and #f
(brk) ⇒ #\t and #f
(brk) ⇒ #\’ and #f
(brk) ⇒ #\s and #f
(brk) ⇒ #\space and #t
(brk) ⇒ #\i and #t
(brk) ⇒ #\t and #f
(brk) ⇒ #\. and #t
(brk) ⇒ #<eof> and #t
\end{verbatim}

It shows the word breaks at those character boundaries shown by the caret ^ below (for clarity, I use _ to indicate the space).

\texttt{T h a t ’ s _ i t .}

^ ^ ^ ^ ^
**make-word-reader** generator return

**make-grapheme-cluster-reader** generator return

{gauche.unicode} The input generator is a generator of characters or codepoints, and return is a procedure that takes a list of characters or codepoints, and returns an object. These procedures create a generator that returns an object at a time, each consists of a word or a grapheme cluster, respectively.

Suppose a generator \( g \) returns characters in a string *That’s it.*, one at a time, again. Then the created generator works as follows:

```scheme
(define brk (make-word-reader g list->string))
(brk)  ⇒  "That’s"
(brk)  ⇒  " "
(brk)  ⇒  "it"
(brk)  ⇒  "."
(brk)  ⇒  #$eof
```

### 9.37.3 Full string case conversion

**string-upcase** string

**string-downcase** string

**string-titlecase** string

**string-foldcase** string

{R6RS|R7RS char|SRFI-129} {gauche.unicode} Converts the case of given string using language-independent full case folding defined by Unicode standard. They differ from SRFI-13’s procedures with the same names (see Section 11.6.8 [SRFI-13 String case mapping], page 698, which simply uses character-by-character case mapping. Notably, the length of resulting string may differ from the source string, and some conversions are sensitive to whether the character is at the word boundary or not. The word boundaries are determined according to UAX #29 text segmentation rules.

```scheme
(string-upcase "straße")
⇒  "STRASSE"
(string-downcase "ΧΑΟΣΧΑΟΣ. ΧΑΟΣ. Σ."
⇒  "χαοσχαοσ. χαοσ. σ."
(string-titlecase "You’re talking about R6RS, right?")
⇒  "You’re Talking About R6rs, Right?"
(string-foldcase "straße")
⇒  "strasse"
(string-foldcase "ΧΑΟΣΣ"
⇒  "χαοσσ"
```

Procedures **string-upcase**, **string-downcase**, and **string-foldcase** are also in R7RS scheme.char module.

Procedure **string-titlecase** is also defined in SRFI-129.

**codepoints-upcase** sequence

**codepoints-downcase** sequence

**codepoints-titlecase** sequence

**codepoints-foldcase** sequence

{gauche.unicode} Like **string-upcase** etc, but these work on a sequence of codepoints instead. Returns a sequence of the same type of the input.

```scheme
(codepoints-upcase '#(115 116 114 97 223 101))
⇒  #(83 84 82 65 83 83 69)
```
**string-ci=?** string1 string2 string3 ...  
**string-ci<?** string1 string2 string3 ...  
**string-ci<=?** string1 string2 string3 ...  
**string-ci>?** string1 string2 string3 ...  
**string-ci=>?** string1 string2 string3 ...  

[R7RS char] \{gauche.unicode\} Case-insensitive string comparison, using full-string case conversion.

Note that Gauche has builtins **string-ci=?** etc., which use character-wise case folding (see Section 6.11.8 [String comparison], page 177). These are different procedures.

\[(string-ci=? \"\u00df\" \"SS\") \Rightarrow \#t\]

### 9.37.4 East asian width property

**char-east-asian-width** char-or-codepoint  
\{gauche.unicode\} The argument may be a character or a nonnegative integer of Unicode codepoint. Returns one of the symbols N (neutral), F (fullwidth), H (halfwidth), W (wide), Na (narrow), and A (ambiguous).

The meaning of this property is explained in Unicode standard annex #11, \url{http://unicode.org/reports/tr11/}.

**string-east-asian-width** str :key F H W Na N A  
\{gauche.unicode\} Computes the ‘width’ of the given string str, taking each character’s East Asian Width int account. It gives you a rough estimate of how much space the string will occupy on the screen, when displayed with monospace fonts.

Although it is true that the exact width is generally undecidable without actually rendering them with glyphs, heuristics can give a good estimate if the string consists of limited scripts. For example, if the string is with ASCII and CJK ideographs, computing ‘Full-width’ and ‘Wide’ letters as twice wide as ASCII character is likely to work.

This procedure assigns the following widths for each character’s East Asian Width category. They can be altered by giving keyword arguments F, H, W, Na, N, and A, respectively:

- F (Full width) 2
- H (Half width) 1
- W (Wide) 2
- Na (Narrow) 1
- N (Neutral) 1
- A (Ambiguous) 2

See UAX #11 (\url{http://unicode.org/reports/tr11/}), section 5, for more detailed discussions of computing the width of displayed strings.

\[(string-east-asian-width "abc") \Rightarrow 3\]  
\[(string-east-asian-width ")\) \Rightarrow 6\]  
\[(string-east-asian-width "1" :W 1.5) \Rightarrow 4.0\]
Chapter 9: Library modules - Gauche extensions

string-take-width str width :key F H W Na N A  [Function]
string-drop-width str width :key F H W Na N A  [Function]

{gauche.unicode} Like string-take and string-drop (see Section 11.6 [String library], page 692), returns a prefix and suffix of the input string str, but using the string's width instead of the number of characters.

string-take-width returns the longest prefix of a string str such that its string-east-asian-width does not exceed width. On the other hand, string-drop-width removes such prefix from str and returns the rest.

The keyword arguments are to customize mappings of East Asian Width categories to numerical widths. See string-east-asian-width above for the details.

9.38 gauche.uvector - Uniform vector library

gauche.uvector  [Module]

Provides procedures that work on uniform vectors (see Section 6.13.2 [Uniform vectors], page 198). This module is a superset of R7RS uniform vector library (scheme.vector.@) and SRFI-4.

The @ part is actually one of u8, s8, u16, s16, u32, s32, u64, s64, f16, f32, f64, c32, c64 or c128.

Gauche’s extension to SRFI-160 is as follows:
• Support of f16vector and c32vector, using 16-bit floating point numbers as used in high-dynamic range image format.
• Efficient element-wise arithmetic procedures, e.g. @vector-add.
• Implements the collection framework (see Section 9.6 [Collection framework], page 404) and the sequence framework (see Section 9.31 [Sequence framework], page 510). So the methods like map, for-each, ref or subseq can be used.
• Some routines takes optional parameters: @vector-ref takes optional fallback value.

When you try to store a number out of the range of the vector type, an error is signaled by default. However, some procedures take an optional argument clamp that specifies alternative behavior in such a case. Clamp argument may take one of the following values.

#f Default behavior (signals an error).
high Clamps high bound; i.e. if the value to be stored is beyond the higher bound of the range, the maximum value is stored instead.
low Clamps low bound; i.e. if the value to be stored is below the lower bound of the range, the minimum value is stored instead.
both Clamps both sides; does both high and low.

(list->u8vector '(1)) ⇒ error
(list->u8vector '(-1) 'low) ⇒ #u8(0)
(list->u8vector '(-1) 'high) ⇒ error
(list->u8vector '(3000) 'high) ⇒ #u8(255)
(list->u8vector '(-100 20 300) 'both) ⇒ #u8(0 20 255)

In the following description, @ can be replaced for any of s8, u8, s16, u16, s32, u32, s64, u64, f16, f32, f64, c32, c64 or c128.

Note: R7RS-large provides separate library for each type, and you should import them individually, for example, (use scheme.vector.u8) (Gauche way) or (import (scheme vector u8)) (R7RS way).

On the other hand, using gauche.uvector imports all the bindings.
9.38.1 Uvector basic operations

The following procedures are built-in; see Section 6.13.2 [Uniform vectors], page 198:

- `make-@vector`
- `uvector?` @vector?
- `uvector-ref` @vector-ref
- `uvector-set!` @vector-set!
- `uvector-length`

[@? obj] [Function] [R7RS vector.@] {gauche.uvector} Returns #t iff obj can be an element of @vector.

[@vector-empty? obj] [Function] [R7RS vector.@] {gauche.uvector} The argument must be a @vector. Returns #t iff it is empty.

[@vector x ...] [Function] [R7RS vector.@] {gauche.uvector} Constructs @vector whose elements are numbers x ... . The numbers must be exact integer for exact integer vectors, and in the valid range of the vector.

```scheme
(s8vector 1 2 3) ⇒ #s8(1 2 3)
```

[-make-uvector class len :optional fill] [Function] [R7RS vector.@] {gauche.uvector} This is a Gauche extension; instead of using separate constructor for each uvector type, you can pass the class of desired uvector.

Type-specific constructors (make-s8vector etc.) are defined in the core library (see Section 6.13.2 [Uniform vectors], page 198).

```scheme
(make-uvector <u8vector> 3) ⇒ #u8(0 0 0)
(make-uvector <s8vector> 5 -1) ⇒ #s8(-1 -1 -1 -1 -1)
```

[-vector-unfold f len seed] [Function] [-vector-unfold-right f len seed] [Function] [R7RS vector.@] {gauche.uvector} Construct a @vector of length len, with each element as a result of (f seed), (f (f seed)), (f (f (f seed))), ... @vector-unfold fills the element from left to right, while @vector-unfold-right from right to left.

```scheme
(u8vector-unfold (cut + 2 <>)) 5 0) ⇒ #u8(2 4 6 8 10)
(u8vector-unfold-right (pa$ + 2) 5 0) ⇒ #u8(10 8 6 4 2)
```

[-vector-unfold! f vec start end seed] [Function] [-vector-unfold-right! f vec start end seed] [Function] [R7RS vector.@] {gauche.uvector} Fill an @vector vec between start-th index (inclusive) and end-th index (exclusive), with the values generated by f, which is called with two arguments, the integer index and the current seed value.

In @vector-unfold!, f is first called with start and seed. It must return two values, the element to put to vec and the next seed value. Then f is called with start + 1 and the previously returned seed value, and so on, until end - start elements are generated. @vector-unfold-right! works similarly, but the elements are generated from right (end-1) to left (start).

If start ≥ end, f is never called and vec isn’t altered. It is an error if the index falls out of range of vec.

Return an unspecified value.
**vector-length vec**

[R7RS vector.@] {gauche.uvector} Returns the length of the `vector` `vec`.

Note that the generic function `size-of` can be used to obtain the length of `vec` as well, if you import `gauche.collection` (see Section 9.6 [Collection framework], page 404).

```scheme```
(s16vector-length '#s16(111 222 333)) ⇒ 3
```

```scheme```
(use gauche.collection)
(size-of '#s16(111 222 333)) ⇒ 3
```

**uvector-size uvector :optional start end**

{gauche.uvector} This function can be applied to any type of uniform vectors, and returns the raw size of the `uvector` in number of octets.

When `start` and/or `end` is/are given, the size of data between those indices are calculated. The special value `-1` for `end` indicates the end of the vector. The returned value matches the number of octets to be written out by `(write-uvector uvector port start end)`.

(Do not confuse this with `vector-length`, which returns the number of elements.)

```scheme```
(uvector-size '#u8(1 2 3)) ⇒ 3
(uvector-size '#u64(1 2 3)) ⇒ 24
```

```scheme```
(uvector-size '#u32(0 1 2 3) 2) ⇒ 8
(uvector-size '#u32(0 1 2 3) 0 1) ⇒ 4
```

**uvector-class-element-size class**

{gauche.uvector} Returns the size of an element of a uvector of the given class, in bytes. An error is raised when `class` is not a uvector class.

```scheme```
(uvector-class-element-size <u8vector>) ⇒ 1
(uvector-class-element-size <s64vector>) ⇒ 8
```

**vector-swap! vec i j**

[R7RS vector.@] {gauche.uvector} Interchanges `i`th and `j`th elements of the uvector `vec`. Return value is not specified.

**vector-fill! vec fill :optional start end**

{gauche.uvector} Stores `fill` in every element of `vec`, ranging from `start` to `end` of `vec`, if they are given. Return value is not specified.

**vector= vec1 ...**

[R7RS vector.@] {gauche.uvector} All arguments must be `vector`s. Returns `#t` iff all arguments have the same length and has the same values (in terms of `=`) at the corresponding position. Zero arguments return `#t`.

Note that in Gauche you can compare uvectors with `equal?` as well.

**vector=? vec1 vec2**

[SRFI-66] {gauche.uvector} Note: This is provided only for the SRFI-66 compatibility. Use `vector=` instead.

Both arguments must be a `vector`. Returns `#t` if `vec1` and `vec2` are equal to each other, `#f` otherwise.

**vector-compare vec1 vec2**

[SRFI-66] {gauche.uvector} Both arguments must be a `vector`. Returns `-1` if `vec1` is smaller than `vec2`, `0` if both are equal to each other, and `1` if `vec1` is greater than `vec2`.

Shorter vector is smaller than longer vectors. If the lengths of both vectors are the same, elements are compared from left to right.
Note that you can compare uvectors with `compare` in Gauche. These are provided because SRFI-66 defines `u8vector-compare`. You can also use them to indicate arguments are vectors of the specific type.

```Scheme
Function @vector-copy vec :optional start end
[R7RS vector.@] {gauche.uvector} Returns a fresh copy of uniform vector vec. If start and/or end are given, they limit the range of vec to be copied.

(u8vector-copy '#u8(1 2 3 4))  ⇒  #u8(1 2 3 4)
(u8vector-copy '#u8(1 2 3 4) 2)  ⇒  #u8(3 4)
(u8vector-copy '#u8(1 2 3 4) 1 3)  ⇒  #u8(2 3)
```

```Scheme
Function uvector-copy vec :optional start end
{gauche.uvector} This is a generic version of @vector-copy. You can give any type of uvector to vec, and get its copy (or copy of its part, depending on start/end argument).
```

```Scheme
Function @vector-reverse-copy vec :optional start end
[R7RS vector.@] {gauche.uvector} Copies vec between strat and end index, but reversing it.

(u8vector-reverse-copy '#u8(1 2 3 4 5))  ⇒  #u8(5 4 3 2 1)

(u8vector-reverse-copy '#u8(1 2 3 4 5) 1 4)  ⇒  #u8(4 3 2)
```

```Scheme
Function @vector-copy! target tstart source :optional sstart send
[R7RS vector.@] {gauche.uvector} Both target and source must be @vectors, and target must be mutable. This procedure copies the elements of source, beginning from index sstart (inclusive) and up to send, into target, beginning from index tstart. sstart and send may be omitted, and in that case 0 and the length of source are assumed, respectively.

(let ((target (u8vector 0 1 2 3 4 5 6)))
  (u8vector-copy! target 2 '#u8(10 11 12 13 14) 1 4) target)
  ⇒  #u8(0 1 11 12 13 6)
```

If the number of elements in the source vector between sstart and send is larger than the target vector beginning from tstart, the excess elements are silently discarded.

It is ok to pass the same vector to target and source; it always works even if the regions of source and destination are overlapping.

*Note:* This procedure used to take just two uniform vectors, target and source, and just copies contents of source to target. Both vectors had to be the same type and same length. The API is revised according to SRFI-160. The old interface is still supported for the backward compatibility, but it is deprecated and will be gone in the future releases.

Also note that SRFI-66 provides `uvector-copy!` with different argument order (see Section 11.16 [Octet vectors], page 720).

```Scheme
Function @vector-reverse! vec :optional start end
[R7RS vector.@] {gauche.uvector}
```

```Scheme
Function @vector-reverse-copy! target tstart source :optional start end
[R7RS vector.@] {gauche.uvector}
```

```Scheme
Function @vector-multi-copy! target tstart tstride source :optional sstart ssize sstride count
{gauche.uvector} This procedure allows different parts of the source uvector source into various parts of the target uvector target, all at once.
```
When ssize is omitted or zero, this procedure does the following:

;;; For each i from 0 to count:
(u8vector-copy! target (+ tstart (* i tstride))
 source sstart)

That is, it copies the content of source (offset by sstart, which defaults to 0) into the target repeatedly, advancing index with tstride. If either the target index reaches the end or count copies are made, the procedure returns. See the example:

(define t (make-u8vector 10 0))
(u8vector-multi-copy! t 0 4 'u8(1 2 3))

\[ \text{t} \Rightarrow \text{#u8(1 2 3 0 1 2 3 0 1 2)} \]

If ssize is given and positive, the source is also splitted as follows:

;;; For each i from 0 to count:
(u8vector-copy! target (+ tstart (* i tstride))
 source (+ sstart (* i sstride))
 (+ sstart (* i sstride) ssize))

That is, each ssize slice from source, is copied into target, advancing source index by sstride and the destination index by dstride. In this case, sstride defaults to ssize if omitted.

(define t (make-u8vector 12 0))
(u8vector-multi-copy! t 0 4 'u8(1 2 3 4 5 6 7 8 9) 2 3)

\[ \text{t} \Rightarrow \text{#u8(1 2 3 0 4 5 6 7 8 9 0)} \]

The operation ends when either count slices are copied, or destination index or source index reaches the end.

Hint: If you want to copy a part of the source vector repeatedly (instead of to its end), you can specify 0 to sstride:

(define t (make-u8vector 12 0))
(u8vector-multi-copy! t 0 4 'u8(1 2 3 4 5 6 7 8 9) 2 4 0)

\[ \text{t} \Rightarrow \text{#u8(3 4 5 6 3 4 5 6 3 4 5 6)} \]

Using collection and sequence framework, you can perform various operations on the homogeneous vectors.

(use gauche.collection)
(use gauche.sequence)

(fold + 0 's32(1 2 3 4)) \Rightarrow 10

(map-to <f32vector> * '#f32(3.2 1.1 4.3) '#f32(-4.3 2.2 9.4))
\Rightarrow #f32(-13.760001 2.420000 40.420002)

(subseq #u32(1 4 3 4 5) 2 4) \Rightarrow #u32(3 4)

uvector-copy! target tstart source :optional sstart send

[Function] {gauche.uvector} This is a generic version of @vector-copy!. The destination target and the source source can be any type of uniform vectors, and they don’t need to match. The copy is done bit-by-bit. So if you copy to a different type of uvector, the result depends on how the numbers are represented internally. This is mainly to manipulate binary data.

Tstart is interpreted according to the type of target, and sstart and send are interpreted according to the type of source.

(rlet1 v (make-u8vector 6 0)
(uvector-copy! v 1 '#u32(0 #x01020304 0) 1 2))
⇒ #u8(0 1 2 3 4 0) or #u8(0 4 3 2 1 0)

@vector-append vec ... [Function]
[R7RS vector.@] {gauche.uvector} All arguments must be @vectors. Returns a fresh vector whose contents are concatenation of the given vectors. (It returns a fresh vector even there’s only one argument).
(u8vector-append '#u8(1 2 3) '#u8(4 5) '#u8() '#u8(6 7 8))
⇒ #u8(1 2 3 4 5 6 7 8)

@vector-concatenate vecs [Function]
[R7RS vector.@] {gauche.uvector} Returns a new @vector which is concatenation of the list of @vectors vecs.
(u8vector-concatenate '(#u8(1 2 3) #u8(4 5 6)))
⇒ #u8(1 2 3 4 5 6)

@vector-append-subvectors :optional vec start end ... [Function]
[R7RS vector.@] {gauche.uvector} Returns a new @vector which is concatenation of the subvectors of given vecs, using accompanied start and end index.
(u8vector-append-subvectors '#u8(1 2 3 4) 1 3 '#u8(5 6 7 8) 0 2)
⇒ #u8(2 3 5 6)

@vector-take vec n [Function]
@vector-take-right vec n [Function]
@vector-drop vec n [Function]
@vector-drop-right vec n [Function]
[R7RS vector.@] {gauche.uvector} Returns a new uvector that has the first n elements, the last n elements, the elements without the first n elements, and the elements without the last n elements, respectively.
N must be a nonnegative exact integer no greater than the length of vec.
(u8vector-take '#u8(0 1 2 3 4 5) 4) ⇒ #u8(0 1 2 3)
(u8vector-drop '#u8(0 1 2 3 4 5) 4) ⇒ #u8(4 5)
(u8vector-take-right '#u8(0 1 2 3 4 5) 4) ⇒ #u8(2 3 4 5)
(u8vector-drop-right '#u8(0 1 2 3 4 5) 4) ⇒ #u8(0 1)
You can also use uvector-alias to obtain a partial uvector that shares the storage with the original uvector, avoiding copying overhead.

@vector-segment vec n [Function]
[R7RS vector.@] {gauche.uvector} Segment vec into fresh uvectors of length n, and returns a list of them. The last uvector may be shorter than n if the length of vec is not a multiple of n.
N must be a positive exact integer.
(u8vector-segment '#u8(0 1 2 3 4 5 6 7) 3)
⇒ ( #u8(0 1 2) #u8(3 4 5) #u8(6 7))

@vector-fold kons knil vec vec2 ... [Function]
@vector-fold-right kons knil vec vec2 ... [Function]
[R7RS vector.@] {gauche.uvector}

@vector-unfold! f vec start end seed [Function]
@vector-unfold-right! f vec start end seed [Function]
[R7RS vector.@] {gauche.uvector}
The optional `start` and `end` arguments limits the portion of `uvector` to search; `start` specifies starting index (inclusive) and `end` specifies ending index (exclusive). Passing `#f` indicates the
default value (0 for start, the length of the vector for end). The returned index is the actual index of the vector, but the elements outside of start-end range don’t need to be sorted.

(uvector-binary-search '#u8(99 99 19 32 58 99) 32 2 5)
⇒ 3

(uvector-binary-search '#u8(99 99 19 32 58 99) 99 2 5)
⇒ #f

The optional skip argument must be a nonnegative exact integer or #f. If it is a positive integer, the number of elements after every key in the uvector is ignored. For example, if skip is 2 and uvector is #u8(3 100 101 5 102 103 13 104 105), only 3, 5 and 13 are subject to search, and elements inbetween are ignored. This allows the caller to store payload, or associated value to each key, in the uvector itself. If skip is positive integer, the length of the searched portion of uvector must be a multiple of the record size (skip+1).

(uvector-binary-search '#u8(3 100 101 5 102 103 13 104 105) 13 #f #f 2)
⇒ 6

(uvector-binary-search '#u8(3 100 101 5 102 103 13 104 105) 13 #f #f 2)
⇒ ; Error: uvector size (8) isn’t multiple of record size (3)

Finally, rounding argument adjusts the behavior when the exact match isn’t found. It can be either one of the following values:

#f This is the default. The procedure searches the element that is equal to key, and returns #f if such element isn’t found.

a symbol floor
When the exact match isn’t found, the procedure returns an index of the element that’s closest to but not greater than key. If key is smaller than all the elements, #f is returned.

a symbol ceiling
When the exact match isn’t found, the procedure returns an index of the element that’s closest to but not smaller than key. If key is greater than all the elements, #f is returned.

(uvector-binary-search '#u32(1 10 100 1000 10000) 3757)
⇒ #f

(uvector-binary-search '#u32(1 10 100 1000 10000) 3757 #f #f #f 'floor)
⇒ 3

(uvector-binary-search '#u32(1 10 100 1000 10000) 3757 #f #f #f 'ceiling)
⇒ 4

Note: SRFI-133 has vector-binary-search, which is quite similar to this procedure (see Section 10.3.2 [R7RS vectors], page 597) but it requires comparison procedure, for it needs to compare general Scheme values. And it does not support skip and rounding arguments.

9.38.2 Uvector conversion operations

@vector->list vec :optional start end
[Function]
[R7RS vector.@] {gauche.uvector} Converts @vector vec to a list. If start and/or end are given, they limit the range of vec to be extracted.

Note that the generic function coerce-to can be used as well, if you import gauche.collection.

(u32vector->list '#u32(9 2 5)) ⇒ (9 2 5)
(use gauche.collection)
(coerce-to <list> ´#u32(9 2 5)) ⇒ (9 2 5)

\texttt{uvector->list} \texttt{ uvec :optional start end}  
\{gauche.uvector\} \texttt{This is a generic version of \texttt{@vector->list}. It can take any kind of \texttt{uvector} as \texttt{uvec}. The meaning of optional arguments are the same as \texttt{@vector->list}.}

\texttt{@vector->vector} \texttt{ vec :optional start end}  
\{R7RS vector.@\} \{gauche.uvector\} \texttt{Converts \texttt{@vector} to a vector. If \texttt{start} and/or \texttt{end} are given, they limit the range of \texttt{vec} to be copied.}

Note that the generic function \texttt{coerce-to} can be used as well, if you import \texttt{gauche.collection}.

(f32vector->vector ´#f32(9.3 2.2 5.5)) ⇒ #(9.3 2.2 5.5)
(f32vector->vector ´#f32(9.3 2.2 5.5) 2) ⇒ #(5.5)

(use gauche.collection)
(coerce-to <vector> ´#f32(9.3 2.2 5.5)) ⇒ #(9.3 2.2 5.5)

\texttt{uvector->vector} \texttt{ uvec :optional start end}  
\{gauche.uvector\} \texttt{This is a generic version of \texttt{@vector->vector}. It can take any kind of \texttt{uvector} as \texttt{uvec}. The meaning of optional arguments are the same as \texttt{@vector->vector}.}

\texttt{list->@vector} \texttt{ list :optional clamp}  
\{R7RS vector.@\} \{gauche.uvector\} \texttt{Converts a list \texttt{list} to a @vector. Optional argument \texttt{clamp} specifies the behavior when the element of \texttt{list} is out of the valid range. (The \texttt{clamp} argument is Gauche’s extension.)}

Note that the generic function \texttt{coerce-to} can be used as well, if you import \texttt{gauche.collection}.

(list->s64vector ´(9 2 5)) ⇒ #s64(9 2 5)
(coerce-to <s64vector> ´(9 2 5)) ⇒ #s64(9 2 5)

\texttt{reverse-list->@vector} \texttt{ list :optional clamp}  
\{R7RS vector.@\} \{gauche.uvector\} \texttt{Create a new @vector with the elements of \texttt{list} in reverse order. Optional argument \texttt{clamp} specifies the behavior when the element of \texttt{list} is out of the valid range. (The \texttt{clamp} argument is Gauche’s extension.)}

\texttt{vector->@vector} \texttt{ vec :optional start end clamp}  
\{R7RS vector.@\} \{gauche.uvector\} \texttt{Converts a vector \texttt{vec} to a @vector. If \texttt{start} and/or \texttt{end} are given, they limit the range of \texttt{vec} to be copied. Optional argument \texttt{clamp} specifies the behavior when the element of \texttt{vec} is out of the valid range. (The \texttt{clamp} argument is Gauche’s extension.)}

Note that the generic function \texttt{coerce-to} can be used as well, if you import \texttt{gauche.collection}.

(vector->f64vector ´#(3.1 5.4 3.2)) ⇒ #f64(3.1 5.4 3.2)
(coerce-to <f64vector> ´#(3.1 5.4 3.2)) ⇒ #f64(3.1 5.4 3.2)
### uvector-alias

**uvector-class vec :optional start end**

{gauche.uvector} This procedure creates an uvector of class `uvector-class` that shares the storage of the given uniform vector `vec`. If optional `start` and `end` arguments are given, only the specified range of `vec` is used for the new vector. Since the storage is shared, modification of the original vector can be seen from the new vector, or vice versa.

The class `uvector-class` must be either one of the uniform vector class, but is not necessary match the class of the source vector `vec`. In such case, the new vector looks at the same region of `vec`’s memory, but interprets it differently. For example, the following code determines whether Gauche is running on big-endian or little-endian machine:

```scheme
(let ((u8v (uvector-alias <u8vector> #u32(1))))
  (if (zero? (u8vector-ref u8v 0))
      'big-endian
      'little-endian))
```

If the `uvector-class` is other than `s8vector` or `u8vector`, the region the new vector points has to meet the alignment requirement. You can assume the beginning of the source vector is aligned suitable for any uniform vectors. So, for example, if you’re creating `u32vector` from `u8vector`, the `start` and `end` must be multiple of 4 (or, if they’re omitted, the length of the original `u8vector` must be multiple of 4). An error is signaled when the given parameters doesn’t satisfy alignment constraint.

### 9.38.3 Uvector numeric operations

These are Gauche extension that allows faster arithmetic over uniform vectors, than extracting and calculating element-wise values.

Most procedures comes with two flavors, a functional version (without `!` in the name) and a linear-update version (with `!` in the name).

A functional version assumes the caller treats the arguments and results immutable objects; mutating them later could have unexpected consequences. (Notably, the functional version may return one of its arguments as is, or returns a pre-computed value, so you shouldn’t assume the return values are freshly allocated objects, unless it is noted so explicitly.)

A linear update version may reuse the storage of the designated argument to produce the return value. Gauche tries to reuse the argument as much as possible, but you should always use the return value and shouldn’t assume the argument itself is modified in-place. In fact, after calling linear-updating procedure, you can’t use the argument that may be modified, since you can’t assume the state of the object after calling the procedure.

#### @vector-add

**@vector-add vec val :optional clamp**

{gauche.uvector} Element-wise arithmetic. Vec must be a `@vector`, and `val` must be either a `@vector`, a vector, or a list of the same length as `vec`, or a number (an exact integer for integer vectors, and a real number for f32- and f64-vectors).

If `val` is a `@vector`, its elements are added to, subtracted from, or multiplied by the corresponding elements of `vec`, respectively, and the results are gathered to a `@vector` and returned. The linear-update version (those have bang `!` in the name) reuses `vec` to store the result, and also returns it. If the result of calculation goes out of the range of `@vector`’s element, the behavior is specified by `clamp` optional argument. (For `f32vector` and `f64vector`, `clamp` argument is ignored and the result may contain infinity).
If \( val \) is a number, it is added to, subtracted from, or multiplied by each element of \( vec \), respectively.

\[
\begin{align*}
(s8vector-add 's8(1 2 3 4) 's8(5 6 7 8)) & \Rightarrow 's8(6 8 10 12) \\
(u8vector-sub 'u8(1 2 3 4) 'u8(2 2 2 2)) & \Rightarrow \text{error} \\
(u8vector-sub 'u8(1 2 3 4) 'u8(2 2 2 2) 'both) & \Rightarrow 'u8(0 0 1 2)
\end{align*}
\]

\[
(f32vector-mul 'f32(3.0 2.0 1.0) 1.5) \Rightarrow 'f32(4.5 3.0 1.5)
\]

@vector-div vec val
@vector-div! vec val
{gauche.uvector} Element-wise division of flonum vectors. These are only defined for f16, f32 and f64vector. \( val \) must be a @vector, a vector or a list of the same length as \( vec \), or a real number.

\[
(f32vector-div 'f32(1.0 2.0 3.0) 2.0) \Rightarrow 'f32(0.5 1.0 1.5)
\]

@vector-and vec val
@vector-and! vec val
@vector-ior vec val
@vector-ior! vec val
@vector-xor vec val
@vector-xor! vec val
{gauche.uvector} Element-wise logical (bitwise) operation. These procedures are only defined for integral vectors. \( val \) must be a @vector, a vector or a list of the same length as \( vec \), or an exact integer. Bitwise and, inclusive or or exclusive or is calculated between each element in \( vec \) and the corresponding element of \( val \) (when \( val \) is a non-scalar value), or \( val \) itself (when \( val \) is an integer). The result is returned in a @vector. The linear-update version reuses \( vec \) to store the result, and also returns it.

@vector-dot vec0 vec1
{gauche.uvector} Calculates the dot product of two @vectors. The length of \( vec0 \) and \( vec1 \) must be the same.

@vector-range-check vec min max
{gauche.uvector} Vec must be a @vector, and each of \( min \) and \( max \) must be either a @vector, a vector or a list of the same length as \( vec \), or a number, or \#f. For each element in \( vec \), this procedure checks if the value is between \( minval \) and \( maxval \) inclusive, where \( minval \) and \( maxval \) are the corresponding values of \( min \) and \( max \) (when \( min \) and/or \( max \) is/are non-scalar value) or \( min \) and \( max \) themselves (when \( min \) and/or \( max \) is/are a number). When \( min \) is \#f, negative infinity is assumed. When \( max \) is \#f, positive infinity is assumed. If all the elements in \( vec \) are within the range, \#f is returned. Otherwise, the index of the leftmost element of \( vec \) that is out of range is returned.

\[
\begin{align*}
(u8vector-range-check 'u8(3 1 0 2) 0 3) & \Rightarrow 'f \\
(u8vector-range-check 'u8(3 1 0 2) 1 3) & \Rightarrow 2 \\
(u8vector-range-check 'u8(4 32 64 98) 0 'u8(10 40 70 90)) & \Rightarrow 3
\end{align*}
\]

;; Range check in a program
(cond
  ((u8vector-range-check u8v 1 31)
   => (lambda (i)
(errorf "\n-sth vector element is out of range: "s 
   i (u8vector-ref u8v i)))
(else (do-something u8v)))

@vector-clamp vec min max
@vector-clamp! vec min max
  {gauche.uvector} Vec must be a @vector, and each of min and max must be either a
  @vector, a vector or a list of the same length as vec, or a number, or #f.

Like @vector-range-check, these procedures check if each element of vec are within the range
between minval and maxval inclusive, which are derived from min and max. If the value is
less than minval, it is replaced by minval. If the value is greater than maxval, it is replaced
by maxval.

@vector-clamp creates a copy of vec and do clamp operation on it, while @vector-clamp!
modifies vec. Both return the clamped vector.

(s8vector-clamp '#s8(8 14 -3 -22 0) -10 10) ⇒ #s8(8 10 -3 -10 0)

9.38.4 Uvector block I/O

A uniform vector can be seen as an abstraction of a chunk of memory. So you might want to
use it for binary I/O. Yes, you can do it.

read-uvector class size :optional iport endian
  {gauche.uvector} Reads size elements of uvector of class class from iport, and returns fleshly
  created uvector. If iport is omitted, the current input port is used.

For example, you can read input as an octet stream as follows:

(with-input-from-string "abcde"
  (^[] (read-uvector <u8vector> 5)))
⇒ #u8(97 98 99 100 101)

If the input port has already reached EOF, an EOF object is returned. The returned uvector
can be shorter than size if the input reaches EOF before size elements are read.

If the iport is a buffered port with ‘modest’ or ‘none’ buffering mode (see Section 6.21.4
[File ports], page 264), read-uvector may return before size elements are read, even if iport
hasn’t reached EOF. The ports connected to a pipe or a network socket behave so by default.
The data is read as a byte stream, so if you give uniform vectors other than s8vector or
u8vector, your result may affected by the endianness. If the optional argument endian
is given, the input is interpreted in that endianness. When omitted, the value of the parameter
default-endian is used. See Section 6.3.7 [Endianness], page 138, for more about endian
handling.

If the size of the input data is unknown and you need to read everything until EOF, use
port->uvector below.

read-bytevector size :optional iport
  {R7RS base} {gauche.uvector} Equivalent to (read-uvector <u8vector> size iport).
This is an R7RS base procedure.

read-uvector! vec :optional iport start end endian
  {gauche.uvector} Reads a chunk of data from the given input port iport, and stores it to
the uniform vector vec. You can give any uniform vector. If optional start and end arguments
are given, they specify the index range in vec that is to be filled, and the rest of the vector
remains untouched. Otherwise, entire vector is used. A special value -1 for end indicates the
end of vec. If iport is omitted, the current input port is used.
If the input reached EOF before the required region of vec is filled, the rest of the vector is untouched.

If iport is already reached EOF when read-uvector! is called, an EOF object is returned. Otherwise, the procedure returns the number of elements read (not bytes).

If the iport is a buffered port with ‘modest’ or ‘none’ buffering mode (see Section 6.21.4 [File ports], page 264), read-uvector! may return before all the elements in vec is filled, even if iport hasn’t reached EOF. The ports connected to a pipe or a network socket behave so by default. If you know there will be enough data arriving and want to make sure vec is filled, change the buffering mode of iport to ‘full’.

The data is read as a byte stream, so if you give uniform vectors other than s8vector or u8vector, your result may affected by the endianness. If the optional argument endian is given, the input is interpreted in that endianness. When omitted, the value of the parameter default-endian is used. See Section 6.3.7 [Endianness], page 138, for more about endian handling.

**read-block! vec :optional iport start end endian**

{gauche.uvector} An old name of read-uvector!. Supported for the backward compatibility, but new code should use read-uvector!.

**port->uvector iport :optional class**

{gauche.uvector} Read data from the input port iport until EOF and store them into a uvector of class. If class is omitted, <u8vector> is used.

If you specify a class of uvector whose element is more than an octet, the input data is packed with platform’s native byteorder.

This procedure is parallel to port->string etc. (see Section 6.21.7.4 [Input utility functions], page 274).

**read-bytevector! bv :optional iport start end**

[R7RS base] {gauche.uvector.} Similar to read-uvector!, but bv must be a u8vector. This is an R7RS base procedure.

**write-uvector vec :optional oport start end endian**

{gauche.uvector} Writes out the content of the uniform vector vec ’as is’ to the output port oport. If oport is omitted, the current output port is used. If optional start and end arguments are given, they specify the index range in vec to be written out. A special value -1 for end indicates the end of vec. This procedure returns an unspecified value.

If you write out a uniform vector except s8vector and u8vector, the care should be taken about the endianness, as in read-uvector!. The optional argument endian specifies the output endian. When it is omitted, the value of the parameter default-endian is used (see Section 6.3.7 [Endianness], page 138).

**write-bytevector bv :optional oport start end**

[Function]

[Function] {gauche.uvector} Similar to write-uvector, but bv must be a u8vector. This is an R7RS base procedure.

**write-block vec :optional oport start end endian**

{gauche.uvector} An old name of write-uvector. Supported for the backward compatibility, but new code should use write-uvector.
9.38.5 Bytevector compatibility

R7RS-small includes bytevectors in its core (scheme.base). In Gauche, bytevectors are the same as u8vectors.

The basic R7RS bytevector procedures are provided in this module. Conversion between bytevectors and strings are provided in gauche.unicode (see Section 9.37.1 [Unicode transfer encodings], page 550) and srfi.181 (see Section 11.40.2 [Transcoded ports], page 761).

bytevector byte . . .
{gauche.uvector} [R7RS base] Alias of u8vector. Returns a fresh bytevector (u8vector) with byte . . . as its elements.

bytevector? obj
{gauche.uvector} [R7RS base] Alias of u8vector?. Returns true iff obj is a bytevector (u8vector).

make-bytevector len :optional byte
{gauche.uvector} [R7RS base] Returns a fresh bytevector (u8vector) of length len. All elements are initialized by byte if given.

R7RS base accepts an exact integer between 0 and 255 inclusive as byte. R7RS bytevector extends the range to -128 to 255 inclusive, while the negative value is wrapped-around by modulo 255. This procedure supports the extended range.

bytevector-length bv
{gauche.uvector} [R7RS base] Alias of u8vector-length. Returns the length of the bytevector (u8vector) bv.

bytevector-u8-ref bv k
bytevector-u8-set! bv k byte
{gauche.uvector} [R7RS base] Alias of u8vector-ref and u8vector-set!. Read and write k-th element of a bytevector (u8vector) bv.

It is an error to give out-of-bound index.

The return value of bytevector-u8-set! is unspecified.

As Gauche’s extension, (setter bytevector-u8-ref) is bytevector-u8-set!.

bytevector-s8-ref bv k
bytevector-s8-set! bv k signed-byte
{gauche.uvector} [R7RS bytevector] Like bytevector-u8-ref and bytevector-u8-set!, but treats octets as a signed byte, ranging from -128 to 127, inclusive.

As Gauche’s extension, (setter bytevector-s8-ref) is bytevector-s8-set!.

bytevector-copy bv :optional start end
{gauche.uvector} [R7RS base] Alias of u8vector-copy. Returns a fresh copy of a bytevector (u8vector) bv. Optionally you can restrict the range of the source vector by indices start (inclusive) and end (exclusive).

bytevector-copy! target tstart source :optional sstart send
{gauche.uvector} [R7RS base] Alias of u8vector-copy!. Both target and source must be bytevectors (u8vectors), and target must be mutable. Copy the content of source (optionally restricting the range between indices start (inclusive) and end (exclusive)) into target starting at the index tstart.
bytevector-copy!-r6 src sstart target tstart len
{gauche.uvector} This is a compatibility procedure for R6RS bytevector-copy! (hence the suffix -r6). When R6RS bytevectors are adopted as R7RS-large scheme.bytevector, the R6RS version of bytevector-copy! comes into R7RS as well (hence R7RS has two different bytevector-copy!, one in scheme.base and one in scheme.bytevector).

It’s unfortunate that R6RS tends to break tradition and invent a new API; here, the arguments differ from other *-copy! procedures: This procedure copies from src, starting from sstart and length len, to target starting tstart.

bytevector-append bv ...
{gauche.uvector} [R7RS base] Alias of u8vector-append. All arguments must be bytevectors (u8vectors). Returns a fresh bytevector whose elements are the concatenation of elements of bv ....

bytevector=? bv1 bv2
{gauche.uvector} [R7RS bytevector] Alias of u8vector=?.. All arguments must be bytevectors (u8vectors). Returns #t iff all bytevectors are of the same size and content.

bytevector->u8-list bv
u8-list->bytevector list
[Function] [Function] [R7RS bytevector] Convert a u8vector to a list of bytes, and vice versa. Same as u8vector->list and list->u8vector, except to not taking optional start/end arguments.

9.39 gauche.version - Comparing version numbers

gauche.version
[Module]

This module provides a convenient procedure to compare version numbers or revision numbers, such as "0.5.1", "3.2-3" or "8.2p11". Usually each release of software component has a version number, and you can define order between them. For example, version "1.2.3" is newer than "1.2" and older than "2.1". You can compare those version numbers like this:

(version<? "2.2.3" "2.2.11") ⇒ #t
(version<? "2.3.1" "2.3") ⇒ #f
(version<? "2.3.1-1" "2.3.1-10") ⇒ #t
(version<? "13a" "5b") ⇒ #f

There are no standard way to name versions, so I chose one convention. This won’t work for all possible variations, but I think it covers typical cases.

Strictly speaking, you can only define partial order between version numbers, for there can be branches. This module uses simple measure and just assumes the version numbers can be fully ordered.

The version number here is defined by the following syntax.

    <version> : <principal-release>
        | <version> <post-subrelease>
        | <version> <pre-subrelease>
    <principal-release> : <relnum>
    <post-subrelease> : [-.] <relnum>
    <pre-subrelease> : _ <relnum>?
    <relnum> : [0-9A-Za-z]+

Typically <relnum> is composed by numeric part and extension part. For example, "23a" is composed by an integer 23 and extension "a". If <relnum> doesn’t begins with digits, we assume its numeric part is -1.
Then, the order of `<relnum>` is defined as follows:

1. If relnum A and relnum B have different numeric part, we ignore the extension and order them numerically, e.g. "3b" < "4a".
2. If relnum A and relnum B have the same numeric part, we compare extension by alphabetically, e.g. "4c" < "4d" and "5" < "5a".

Given the order of `<relnum>`, the order of version numbers are defined as follows:

1. Decompose each version number into a list of `<principal-release>` and subsequence subrelease components. We call each element of the list "release components".
2. If the first release component of both lists are the same, remove it from both. Repeat this until the head of the lists differ.
3. Now we have the following cases.
   1. Both lists are empty: versions are the same.
   2. One list (A) is empty and the other list (B) has post-subrelease at head: A is prior to B
   3. One list (A) is empty and the other list (B) has pre-subrelease at head: B is prior to A
   4. List A’s head is post-subrelease and list B’s head is pre-subrelease: B is prior to A
   5. Both lists have post-subrelease or pre-subrelease at head: compare their relnums.

Here are some examples:

"1" < "1.0" < "1.1" < "1.1.1" < "1.1.2" < "1.2" < "1.11"
"1.2.3" < "1.2.3-1" < "1.2.4"
"1.2.3" < "1.2.3a" < "1.2.3b"
"1.2_" < "1.2_rc0" < "1.2_rc1" < "1.2" < "1.2-pl1" < "1.2-pl2"
"1.1-patch112" < "1.2_alpha"

The reason of having `<pre-subrelease>` is to allow "release candidate" or "pre-release" version.

A trick: If you want “version 1.2 release or later", you can say (version<=? "1.2" v). This excludes prerelease versions such as 1.2_pre3. If you want “version 1.2 release or later”, you can say (version<=? "1.2_" v), which includes 1.2_pre1 etc., but excludes anything below, such as 1.1.99999.

It is common if you want to specify acceptable versions with combination of conditions, e.g. “version 1.3 or later, except version 1.4.1” or “greater than version 1.1 and below 1.5”. A version spec is an S-expression to represent that condition. You can use version-satisfy? to check if given version satisfies the spec.

The syntax of version spec is as follows.

```s-expression
<version-spec> : <version>
   | (op <version-spec>)
   | (and <version-spec> ...)
   | (or <version-spec> ...)
   | (not <version-spec>)
<version> : version string
<op> : = | < | <= | > | >=
```

version=? ver1 ver2
version<? ver1 ver2
version<=? ver1 ver2
version>? ver1 ver2
version>=? ver1 ver2
    \{gauche.version\} Returns a boolean value depending on the order of two version number
string ver1 and ver2. If the arguments contain invalid strings as the defined version number,
an error is signaled.

version-compare ver1 ver2
    \{gauche.version\} Compares two version number strings ver1 and ver2, and returns either
-1, 0, or 1, depending whether ver1 is prior to ver2, ver1 is the same as ver2, or ver1 is after
ver2, respectively.

relnum-compare rel1 rel2
    \{gauche.version\} This is lower-level procedure of version-compare. Compares two release
numbers (relnums) rel1 and rel2, and returns either -1, 0, or 1 depending whether rel1 is
prior to rel2, rel1 is the same as rel2, or rel1 is after rel2, respectively.

The following procedures are to check if a given version satisfies a version specification.

valid-version-spec? spec
    \{gauche.version\} This is a syntax checker. Returns #t if spec is a valid version specifi-
cation, #f otherwise. See gauche.version module description for the definition of version
specification.

version-satisfy? spec version
    \{gauche.version\} Returns #t if version satisfies a version specification spec, #f otherwise.
See gauche.version module description for the definition of version specification.

9.40 gauche.vport - Virtual ports

gauche.vport
    \{Module\}
    Virtual ports, or procedural ports, are the ports whose behavior can be programmed in
Scheme.

This module provides two kinds of virtual ports: Fully virtual ports, in which every I/O op-
eration invokes user-provided procedures, and virtual buffered ports, in which I/O operations
are done on an internal buffer and user-provided procedures are called only when the buffer
needs to be filled or flushed.

This module also provides virtual buffered ports backed up by a uniform vector, as an example
of the feature.

Fully virtual ports

This type of virtual ports are realized by classes <virtual-input-port> and <virtual-output-
port>. You can customize the port behavior by setting appropriate slots with procedures.

<virtual-input-port>
    \{gauche.vport\} An instance of this class can be used as an input port. The behavior of the
port depends on the settings of the instance slot values.

To work as a meaningful input port, at least either one of getb or getc slot must be set. Otherwise,
the port returns EOF for all input requests.

getb
    \{Instance Variable of <virtual-input-port>\}
    If set, the value must be a procedure that takes no arguments. Every time binary input
is required, the procedure is called.

    The procedure must return an exact integer between 0 and 255 inclusive, or #f or an EOF
object. If it returns an integer, it becomes the value read from the port. If it returns other
values, the port returns EOF.
If the port is requested a character input and it doesn’t have the `getc` procedure, the port calls this procedure, possibly multiple times, to construct a whole character.

**getc**  
[Instance Variable of `<virtual-input-port>`]  
If set, the value must be a procedure that takes no arguments. Every time character input is required, the procedure is called.

The procedure must return a character, `#f` or an EOF object. If it returns a character, it becomes the value read from the port. If it returns other values, the port returns EOF.

If the port is requested a binary input and it doesn’t have the `getb` procedure, the port calls this procedure, then converts a character into a byte sequence, and use it as the binary value(s) read from the port.

**gets**  
[Instance Variable of `<virtual-input-port>`]  
If set, the value must be a procedure that takes one argument, a positive exact integer. It is called when the block binary input, such as `read-uvector`, is requested. It must return a (maybe incomplete) string up to the specified size, or `#f` or EOF object. If it returns a null string, `#f` or EOF object, the port thinks it reached EOF. If it returns other string, it is used as the result of block read. It shouldn’t return a string larger than the given size (Note: you must count size (bytes), not the number of characters). The reason of this procedure is efficiency; if this procedure is not provided, the port calls `getb` procedure repeatedly to prepare the block of data. In some cases, providing block input can be much more efficient (e.g. suppose you’re reading from a block of memory chunk).

You can leave this slot unset if you don’t need to take such advantage.

**ready**  
[Instance Variable of `<virtual-input-port>`]  
If set, the value must be a procedure that takes one boolean argument. It is called when `char-ready?` or `byte-ready?` is called on the port. The value returned from your procedure will be the result of these procedures.

The boolean argument is `#t` if `char-ready?` is called, or `#f` if `byte-ready?` is called.

If unset, `char-ready?` and `byte-ready?` always return `#t` on the port

**close**  
[Instance Variable of `<virtual-input-port>`]  
If set, the value must be a procedure that takes no arguments. It is called when the port is closed. Return value is discarded. You can leave this unset if you don’t need to take an action when the port is closed.

This procedure may be called from a finalizer, so you have to be careful to write it. See the note on finalization below.

**seek**  
[Instance Variable of `<virtual-input-port>`]  
If set, the value must be a procedure that takes two arguments, offset and whence. The meaning of them is the same as the arguments to `port-seek` (see Section 6.21.3 [Common port operations], page 260). The procedure must adjust the port’s internal read pointer so that the next read begins from the new pointer. It should return the updated pointer (the byte offset from the beginning of the port).

If unset, call of `port-seek` and `port-tell` on this port will return `#f`.

Note that this procedure may be called for the purpose of merely querying the current position, with 0 as `offset` and `SEEK_CUR` as `whence`. If your port knows the read pointer but cannot move it, you can still provide this procedure, which returns the current pointer position for such queries and returns `#f` for other arguments.
<virtual-output-port>

{gauche.vport} An instance of this class can be used as an output port. The behavior of
the port depends on the settings of the instance slot values.

To work as an output port, at least either one of putb or putc slot has to be set.

putb

[Instance Variable of <virtual-output-port>]
If set, the value must be a procedure that takes one argument, a byte value (exact integer
between 0 and 255, inclusive). Every time binary output is required, the procedure is
called. The return value of the procedure is ignored.

If this slot is not set and binary output is requested, the port may signal an <io-unit-
error> error.

putc

[Instance Variable of <virtual-output-port>]
If set, the value must be a procedure that takes one argument, a character. Every time
character output is required, the procedure is called. The return value of the procedure is
ignored.

If this slot is not set but putb slot is set, the virtual port decomposes the character into
a sequence of bytes then calls putb procedures.

puts

[Instance Variable of <virtual-output-port>]
If set, the value must be a procedure that takes a (possibly incomplete) string. The return
value of the procedure is ignored.

This is for efficiency. If this slot is not set, the virtual port calls putb or putc repeatedly
to output a chunk of data. But if your code can perform chunked output efficiently, you
can provide this procedure.

flush

[Instance Variable of <virtual-output-port>]
If set, the value must be a procedure that takes no arguments. It is called when flushing
a port is required (e.g. flush is called on the port, or the port is being closed).

This procedure is useful that your port does some sort of buffering, or needs to keep some
state. If your port doesn’t do stateful operation, you can leave this unset.

This procedure may be called from a finalizer, and needs a special care. See notes on
finalizers below.

close

[Instance Variable of <virtual-output-port>]
The same as <virtual-input-port>’s close slot.

seek

[Instance Variable of <virtual-output-port>]
The same as <virtual-input-port>’s seek slot.

Virtual buffered ports

This type of virtual ports are realized by classes <buffered-input-port> and
<buffered-output-port>. You can customize the port behavior by setting appropriate slots
with procedures.

Those ports have internal buffer and only calls Scheme procedures when the buffer needs to
be filled or flushed. Generally it is far more efficient than calling Scheme procedures for every
I/O operation. Actually, the internal buffering mechanism is the same as Gauche’s file I/O
ports.

These ports uses u8vector as a buffer. See Section 6.13.2 [Uniform vectors], page 198, for
the details.
<buffered-input-port>  
{gauche.vport} An instance of this class behaves as an input port. It has the following instance slots. For a meaningful input port, you have to set at least fill slot.

fill  
[Instance Variable of <buffered-input-port>]  
If set, it must be a procedure that takes one argument, a u8vector. It must fill the data from the beginning of the vector. It doesn’t need to fill the entire vector if there’s not so many data. However, if there are remaining data, it must fill at least one byte; if the data isn’t readily available, it has to wait until some data becomes available.

The procedure must return a number of bytes it actually filled. It may return 0 or an EOF object to indicate the port has reached EOF.

ready  
[Instance Variable of <buffered-input-port>]  
If set, it must be a procedure that takes no arguments. The procedure must return a true value if there are some data readily available to read, or #f otherwise. Unlike fully virtual ports, you don’t need to distinguish binary and character I/O.

If this slot is not set, the port is regarded as it always has data ready.

close  
[Instance Variable of <buffered-input-port>]  
If set, it must be a procedure that takes no arguments. The procedure is called when the virtual buffered port is closed. You don’t need to set this slot unless you need some cleaning up when the port is closed.

This procedure may be called from a finalizer, and needs special care. See the note on finalization below.

filenum  
[Instance Variable of <buffered-input-port>]  
If set, it must be a procedure that returns underlying file descriptor number (exact non-negative integer). The procedure is called when port-file-number is called on the port. If there’s no such underlying file descriptor, you can return #f, or you can leave this slot unset.

seek  
[Instance Variable of <buffered-input-port>]  
If set, it must be a procedure that takes two arguments, offset and whence. It works the same way as <virtual-input-port>’s seek procedure; see above.

This procedure may be called from a finalizer, and needs special care. See the note on finalization below.

Besides those slot values, you can pass an exact nonnegative integer as the :buffer-size keyword argument to the make method to set the size of the port’s internal buffer. If :buffer-size is omitted, or zero is passed, the system’s default buffer size (something like 8K) is used. :buffer-size is not an instance slot and you cannot set it after the instance of the buffered port is created. The following example specifies the buffered port to use a buffer of size 64K:

(make <buffered-input-port> :buffer-size 65536 :fill my-filler)

<buffered-output-port>  
{gauche.vport} An instance of this class behaves as an output port. It has the following instance slots. You have to set at least flush slot.

flush  
[Instance Variable of <buffered-output-port>]  
If set, it must be a procedure that takes two arguments, an u8vector buffer and a flag. The procedure must output data in the buffer to somewhere, and returns the number of bytes actually output.

If the flag is false, the procedure may output less than entire buffer (but at least one byte). If the flag is true, the procedure must output entire buffer.
close  [Instance Variable of <buffered-output-port>]
    Same as <buffered-input-port>’s close slot.

filenum  [Instance Variable of <buffered-output-port>]
    Same as <buffered-input-port>’s filenum slot.

seek  [Instance Variable of <buffered-output-port>]
    Same as <buffered-input-port>’s seek slot.

Besides those slot values, you can pass an exact nonnegative integer as the :buffer-size keyword argument to the make method to set the size of the port’s internal buffer. See the description of <buffered-input-port> above for the details.

Uniform vector ports

The following two procedures return a buffered input/output port backed up by a uniform vector. The source or destination vector can be any type of uniform vector, but they will be aliased to u8vector (see uvector-alias in Section 9.38.2 [Uvector conversion operations], page 563).

If used together with pack/unpack (see Section 12.2 [Packing binary data], page 811), it is useful to parse or construct binary data structure. It is also an example of using virtual ports; read gauche/vport.scm (or ext/vport/vport.scm in the source tree) if you’re curious about the implementation.

open-input-uvector uvector  
    {gauche.vport} Returns an input port that reads the content of the given uniform vector uvector from its beginning. If reading operation reaches the end of uvector, EOF is returned. Seek operation is also implemented.

open-input-bytevector u8vector  
    [R7RS base] {gauche.vport} Similar to open-input-uvector, but the argument must be an u8vector. This is an R7RS base procedure.

open-output-uvector :optional uvector :key extendable  
    {gauche.vport} Returns an output port that uses the given uvector as the storage for the data output to the port.

    If uvector is completely filled, what happens after that depends on extendable - if it is false (default), the rest of data is discarded silently. If it is true, the storage is extended automatically to accommodate more data.

    If you give true value to extendable, you have to retrieve the result by get-output-uvector below, since the uvector you passed in won’t contain spilled data.

    As a special case, you can omit uvector argument; then u8vector is used as the storage. In that case you can’t specify extendable keyword argument, but it is assumed true, since it won’t make sense otherwise. Use get-output-uvector to retrieve the stored result.

    Seek operation is also implemented. Note that the meaning of SEEK_END whence differ between extendable and fixed-size uvector ports. For extendable ports, the end whence placed next to the biggest offset of the data ever written; if you open a port and just write one byte, the end whence is the second byte, no matter how big the existing buffer is. On the other hand, for fixed-size uvector ports, end whence is fixed to the next to the end of the given buffer, no matter how much data you’ve written to it. In the latter case, you can’t seek on or past the end (you need to pass negative number along SEEK_END to port-seek).

open-output-bytevector  
    [R7RS base] {gauche.vport} Same as open-output-uvector without arguments. Uses extensible u8vector as the buffer. This is an R7RS base procedure.
The returned uvector is the same type as the one passed to `open-output-uvector`, containing up to actually written data; it may be smaller than the uvector passed to `open-output-uvector`; it can be larger if the port is extendable.

If the type of uvector is other than `s8vector` and `u8vector`, and the written data doesn’t fill up the whole element won’t be in the result. For example, if you use `s32vector` to create the port, then write 7 bytes to it, `get-output-uvector` returns a single element `s32vector`, for the last 3 bytes does not consist a whole 32bit integer.

By default, the returned vector is a fresh copy of the contents. Passing true value to `shared` may avoid copying and allow sharing storage for the one being used by `port`. If you do so, keep in mind that if you seek back and write to `port` subsequently, the content of returned vector may be changed.

This is an R7RS base procedure.

List ports

The following procedures allow you to use list of characters or octets as a source of an input port. These are (a kind of) opposite of `port->list` family (see Section 6.21.7.4 [Input utility functions], page 274) or `port->char-lseq` family (see Section 6.18.2 [Lazy sequences], page 241).

`open-input-char-list char-list` [Function]

`open-input-byte-list byte-list` [Function]

A caveat: Gauche allows mixing binary input and textual input from the same port. If you read or even peek a byte from a port created from a character list, the port buffers a character and disassembles it to bytes; the disassembled character may not be included in the remaining input list.

Generator ports

The following procedures allow you to use character generators or byte generators as a source of an input port. These are (a kind of) opposite of `port->char-generator` family (see Section 9.12.1 [Generator constructors], page 436).

`open-input-char-generator cgen` [Function]

`open-input-byte-generator bgen` [Function]
must be a generator that yields bytes (exact integers between 0 and 255, inclusive). An error
will be raised if the given generator yields incorrect type of objects.

\[
\text{(read (open-input-char-generator (string->generator "foo")))}
\Rightarrow \text{foo}
\]

Since the generators are objects relying on side effects, you shouldn’t use cgen or bgen after
you pass them to those procedures; if you use them afterwards, the result is undefined.

get-remaining-input-generator \text{port} \quad \text{[Function]}
{gauche.vport} If \text{port} is the one created by open-input-char-generator or open-input-
byte-generator, returns a generator that yields the characters or bytes that haven’t been
read yet. If the port already read everything, an empty generator is returned.

Once you take the remaining input generator, you should no longer read from the input
generator ports; they share internal states and mixing them will likely to cause unexpected
behaviors. If side-effects safe behavior is desired, use lazy sequence and input list ports.

Accumulator ports
Accumulators are dual to generators; it’s a procedure that accepts a value at a time, and the
end of the value is indicated by EOF. See Section 10.3.12 [R7RS generators], page 631, for the
basic operations of accumulators.

The following procedures turns an accumulator that accepts characters or octets into an
output port.

open-output-char-accumulator \text{acc} \quad \text{[Function]}
{gauche.vport} Returns an output port that sends the output characters to an accumulator
\text{acc}, which takes a character as the argument. When the returned output port is closed, EOF
is passed to \text{acc}.

Note: The behavior is undefined if you try to perform binary output to the returned output
port.

open-output-char-accumulator \text{acc} \quad \text{[Function]}
{gauche.vport} Returns an output port that sends the output bytes to an accumulator \text{acc},
which takes a byte as the argument. When the returned output port is closed, EOF is passed
to \text{acc}.

A character sent to the output port is converted to octets in the Gauche’s native encoding.

open-output-accumulator \text{acc} \quad \text{[Function]}
{gauche.vport} The accumulator \text{acc} must accept a byte, a character or a string. Returns
an output port that sends the output data to the \text{acc}. When the returned output port is
closed, EOF is passed to \text{acc}.

Note on finalization
If an unclosed virtual port is garbage collected, its close procedure is called (in case of virtual
buffered ports, its flush procedure may also be called before close procedure). It is done by
a finalizer of the port. Since it is a part of garbage-collection process (although the Scheme
procedure itself is called outside of the garbage collector main part), it requires special care.

- It is possible that the object the virtual port has a reference may already be finalized. For
  example, if a virtual port \text{X} holds the only reference to a sink \text{Y}, to which the output
goes. \text{X}'s flush procedure sends its output to \text{Y}. However, if flush procedure can be
called from a finalizer, it may be possible that \text{Y}'s finalizer has already been called and \text{Y}
is closed. So \text{X}'s flush procedure has to check if \text{Y} has not been closed.
• You cannot know when and in which thread the finalizer runs. So if the procedure like `close` or `flush` of virtual ports need to lock or access the global resource, it needs to take extra care of avoiding dead lock or conflict of access.

Even in single thread programs, the finalizer can run anywhere in Scheme programs, so effectively it should be considered as running in a different thread.
10 Library modules - R7RS standard libraries

Gauche predates R7RS, and for the convenience, Gauche makes quite a few procedures as built-in (see Chapter 6 [Core library], page 105). Although the set of Gauche’s core features are mostly superset of R7RS, some functions and syntaxes have different names and/or interface from R7RS.

R7RS fully-compatible syntaxes and functions are available in the set of modules described in this chapter. Since R7RS programs and libraries needs to follow a specific format (import declaration or define-library form), generally there’s no ambiguity in whether you’re looking at R7RS code or Gauche-specific code. Also, it is totally transparent to load R7RS library into Gauche-specific code or vice versa. However, you need to be aware of which “world” you’re in when you code.

If you’re familiar with Gauche, take a look at the section Section 10.1 [R7RS integration], page 580, which describes how you can go back and forth between Gauche and R7RS.

10.1 R7RS integration

10.1.1 Traveling between two worlds back and forth

When you start Gauche, either in REPL or as a script, you’re in user module, which inherits gauche module. Likewise, when you read a library, the initial module inherits gauche module (until you call select-module). That’s why you can access all the built-in procedures of Gauche without saying (use something). (See Section 4.13.5 [Module inheritance], page 83, for the details about inheriting modules).

On the other hand, R7RS requires to be explicit about which namespaces you’ll be using, by import form, e.g. (import (scheme base)). Besides, R7RS library must be explicitly enclosed by define-library form. Before the first import form of a program, or outside of define-library, is beyond R7RS world—the standard defines nothings about it.

These facts let Gauche to set up appropriate “world”, and you can use R7RS code and traditional Gauche code transparently.

NB: As explained in Section 10.1.2 [Three forms of import], page 582, R7RS import is rather different from Gauche import, so we note the former r7rs#import and the latter gauche#import in this section for clarity. When you write code don’t use prefixes r7rs# and gauche#; just write import.

Loading R7RS libraries

The define-library form is defined as a macro in gauche module; it sets up R7RS environment before evaluating its contents. So, when you load an R7RS library (either from Gauche code via use form, or from R7RS code via r7rs#import form), Gauche starts loading the file in gauche module, but immediately see define-library form, and the rest is handled in R7RS environment.

Suppose you have an R7RS library (mylib foo) with the following code:

```
(define-library (mylib foo)
  (import (scheme base))
  (export snoc)
  (begin
    (define (snoc x y) (cons y x))))
```

It should be saved as mylib/foo.scm in one of the directories in *load-path*.

From R7RS code, this library can be loaded by r7rs#import:

```
(import (mylib foo))
```
To use this library from Gauche code, concatenate elements of library names by . to get a module name, and use it:

```gauche
(use mylib.foo)
```

```
(snoc 1 2) ⇒ (2 . 1)
```

### Loading Gauche libraries
To use Gauche library foo.bar from R7RS code, split the module name by . to make a list for the name of the library. For example, `gauche.lazy` module can be used from R7RS as follows:

```gauche
(import (gauche lazy))
```

For SRFI modules, R7RS implementations have a convention to name it as `(srfi n)`, and Gauche follows it. The following code loads SRFI-1 and SRFI-13 from R7RS code:

```gauche
(import (srfi 1) (srfi 13))
```

(It’s not that Gauche treat `srfi` name specially; installation of Gauche includes adapter libraries such as `srfi/1.scm`.)

A tip: To use Gauche’s built-in features (the bindings that are available by default in Gauche code) from R7RS code, import `gauche/base` library (see Section 9.2 [Importing gauche built-ins], page 373):

```gauche
(import (gauche base))
```

```
filter ⇒ #<closure filter>
```

### Running R7RS scripts
R7RS scripts always begin with `import` form. However, `r7rs/import` has a different syntax and semantics from `gauche/import`—so we employ a trick.

When `gosh` is started, it loads the given script file in `user` module. We have a separate `user/import` macro, which examines its arguments and if it is R7RS import syntax, switch to the `r7rs.user` module and run the `r7rs/import`. Otherwise, it runs `gauche/import`. See Section 10.1.2 [Three forms of import], page 582, for the details.

An example of R7RS script:

```r7rs
(import (scheme base) (scheme write))
(display "Hello, world!\n")
```

If you’re already familiar with Gauche scripts, keep in mind that R7RS program doesn’t treat `main` procedure specially; it just evaluates toplevel forms from top to bottom. So the following script doesn’t output anything:

```r7rs
(import (scheme base) (scheme write))
(define (main args)
  (display "Hello, world!\n")
  0)
```

To access the command-line arguments in R7RS scripts, use `command-line` in `(scheme process-context)` library (see Section 10.2.12 [R7RS process context], page 590, also see Section 6.24.2 [Command-line arguments], page 294).

### Using R7RS REPL
When `gosh` is invoked with `-r7` option and no script file is given, it enters an R7RS REPL mode. For the convenience, the following modules (“libraries”, in R7RS term) are pre-loaded.

```r7rs
(scheme base) (scheme case-lambda) (scheme char)
```
Besides, the history variables \texttt{*1}, \texttt{*2}, \texttt{*3}, \texttt{*1+}, \texttt{*2+}, \texttt{*3+}, \texttt{*e} and \texttt{*history} are available (See Section 3.2.1 [Working in REPL], page 24, for the details of history variables).

You can know you're in R7RS REPL by looking at the prompt, where \texttt{gosh} shows the current module \texttt{(r7rs.user)}:

\begin{verbatim}
  gosh[r7rs.user]>
\end{verbatim}

To switch Gauche REPL from R7RS REPL, import \texttt{(gauche base)} and select \texttt{user} module using \texttt{select-module}:

\begin{verbatim}
  gosh[r7rs.user]>(import (gauche base))
  #<undef>
  gosh[r7rs.user]>(select-module user)
  #<undef>
  gosh>
\end{verbatim}

(You can \texttt{(select-module gauche)} but that's usually not what you want to do—changing \texttt{gauche} module can have unwanted side effects.)

When you’re working on R7RS code in file and load it into R7RS REPL (for example, if you’re using Emacs Scheme mode, C-c C-1 does the job), make sure the file is in proper shape as R7RS: that is, the file must start with appropriate \texttt{import} declarations, or the file contains \texttt{define-library} form(s). If you load file without those forms, it is loaded into Gauche’s \texttt{user} module no matter what your REPL’s current module is, and the definitions won’t be visible from \texttt{r7rs.user} module by default.

\section*{Switching from Gauche REPL}

By default, \texttt{gosh} enters Gauche REPL when no script file is given. See Section 3.2.1 [Working in REPL], page 24, for detailed explanation of using REPL.

To switch Gauche REPL to R7RS REPL, simply use \texttt{r7rs-style} import; \texttt{user#import} knows you want R7RS and make a switch:

\begin{verbatim}
  gosh> (import (scheme base))
  #<undef>
  gosh[r7rs.user]>
\end{verbatim}

If you don’t start \texttt{gosh} with \texttt{-r7} option, however, only the libraries you given to \texttt{user#import} are loaded at this moment.

If you want to switch the “vanilla” \texttt{r7rs} environment, that is, even not loading \texttt{(scheme base)}, then you can use \texttt{r7rs} module and directly select \texttt{r7rs.user}:

\begin{verbatim}
  gosh> (use r7rs)
  gosh> (select-module r7rs.user)
  gosh[r7rs.user]>
\end{verbatim}

If you do this, the only bindings visible initially are \texttt{import} and \texttt{define-library}; even \texttt{define} is undefined! You have to manually do \texttt{(import (scheme base))} etc. to start writing Scheme in this environment.

\subsection*{10.1.2 Three import forms}

For historical reasons, Gauche has three \texttt{import} forms; the original Gauche’s \texttt{import}, R7RS \texttt{import}, and the hybrid \texttt{import}.
Usually it is clear that the code is written in traditional Gauche or in R7RS, and usage of import is typically idiomatic, so there’s not much confusion in practice. Only when you talk about import outside of code, you might need to specify which one you’re talking.

The hybrid import is what we described user#import in the previous section (see Section 10.1.1 [Traveling between two worlds back and forth], page 580). It understands both of Gauche’s import and R7RS import. So what you really need to know is the first two.

Gauche’s module system design is inherited from STk, and we’ve been used import for purely name-space level operation; that is, it assumes the module you import from already exists in memory. Loading a file that defines the module (if necessary) is done by separate primitives, require. In most cases one file defines one module, and using that module means require it then import it (it’s so common that Gauche has a macro for it—use). However, separating those two sometimes comes handy when you need some nontrivial hacks. See Section 4.13.4 [Using modules], page 81, for the details of Gauche’s import.

R7RS leaves out the relation between modules (libraries) and files in order to give implementation freedom. If necessary, its import must load a file implicitly and transparently. So R7RS’s import is semantically Gauche’s use.

The hybrid import only appears at the beginning of the Scheme scripts. It finds out whether the script is in the traditional Gauche code or in the R7RS code. See Section 10.1.1 [Traveling between two worlds back and forth], page 580, for the details.

Now we’ll explain R7RS import:

```
import import-spec ... [Special Form]

[R7RS] Imports libraries specified by import-specs. What R7RS calls libraries are what Gauche calls modules; they’re the same thing.

R7RS libraries are named by a list of symbols or integers, e.g. (scheme base) or (srfi 1). It is translated to Gauche’s module name by joining the symbols by periods; so, R7RS (scheme base) is Gauche’s scheme.base. Conversely, Gauche’s data.queue is available as (data queue) in R7RS. To use those two libraries, R7RS program needs this form at the beginning.

(import (scheme base)
  (data queue))

It works just like Gauche’s use forms; that is, if the named module doesn’t exist in the current process, it loads the file; then the module’s exported bindings become visible from the current module.

(use scheme.base)
(use data.queue)

(You may wonder what if R7RS library uses symbols with periods in them. Frankly, we haven’t decided yet. It’ll likely be that we use some escaping mechanism; for the time being you’d want to stick with alphanumeric characters and hyphens as possible.)

Just like Gauche’s use, you can select which symbols to be imported (or not imported), rename specific symbols, or add prefix to all imported symbols. The formal syntax of R7RS import syntax is as follows:

```
<import declaration> : (import <import-set> <import-set> ...)

<import-set> : <library-name>
  | (only <import-set> <identifier> <identifier> ...)
  | (except <import-set> <identifier> <identifier> ...)
  | (prefix <import-set> <identifier>)
  | (rename <import-set>
    <identifier> <identifier>)
    <identifier> <identifier>) ...
```
10.2 R7RS small language

10.2.1 R7RS library form

R7RS libraries are defined by define-library form.

In R7RS view, define-library form itself does not belong to a Scheme code—it exists outside of the Scheme world. It defines the boundary of R7RS Scheme; inside define-library there is R7RS world, but outside, it’s not a business of R7RS. For example, you can’t generate define-library by a macro, within R7RS specification.

In Gauche, we implement R7RS world inside Gauche world; define-library itself is interpreted in the Gauche world. In fact, define-library is a Gauche macro. However, if you’re writing portable R7RS code, you should forget how define-library is implemented, and do not put anything outside of define-library form.

\[\text{define-library library-name library-decl ...} \]

[Macro]

[R7RS] Defines a library library-name, which is a list of symbols or base-10 integer:

\[<\text{library-name}> : (<\text{identifier-or-base-10-integer}> <\text{identifier-or-base-10-integer}> ...)\]

Library declarations library-decl can be export declarations, import declarations, begin-list of Scheme code, include forms, or cond-expand forms.

\[<\text{library-decl}> : (\text{export <export-spec> ...}) | (\text{import declaration}) | (\text{begin <command-or-definition> ...}) | (\text{include <string> <string2> ...}) | (\text{include-ci <string> <string2> ...}) | (\text{include-library-declarations <string> <string2> ...}) | (\text{cond-expand <cond-expand-clause} <cond-expand-clause2> ...) | (\text{cond-expand <cond-expand-clause} <cond-expand-clause2> ... (else <library-decl> ...))\]

The export declaration is the same Gauche’s export form; see Section 4.13.4 [Using modules], page 81.

The import declaration is R7RS’s import form, described in Section 10.1.2 [Three forms of import], page 582.

The include and include-ci declarations are the same as Gauche’s; see Section 4.11 [Inclusions], page 74. Note that Gauche allows any code to be included—the content of the named file is simply wrapped with begin and substituted with these forms—but in R7RS definition, what you include must contain only Scheme code (not one of the library declarations or define-library form).

The include-library-declarations declaration works like include, but the content of the read file is interpreted as library declarations instead of Scheme code.

The cond-expand declaration is also the same as Gauche’s; see Section 4.12 [Feature conditional], page 75. When used directly below define-library, it must expands to one of the library declarations.
10.2.2 scheme.base - R7RS base library

scheme.base  [Module]
Exports bindings of R7RS (scheme base) library. From R7RS programs, those bindings are available by (import (scheme base)).

Bindings common to Gauche’s built-Ins
The following syntaxes and procedures are the same as Gauche’s builtins:

Primitive expression types
quote if include include-ci lambda

Derived expression types
cond case and or when unless cond-expand let let* letrec letrec* let-values let*-values begin do make-parameter parameterize guard quasiquote unquote unquote-splicing case-lambda

Macros
let-syntax letrec-syntax syntax-rules syntax-error define-syntax

Variable definitions
define define-values

Record type definitions
define-record-type

Equivalence predicates
eqv? eq? equal?

Numbers
= < > <= > = zero? positive? negative? odd? even? max min + * - / abs floor/ floor-quotient floor-remainder truncate/ truncate-quotient truncate-remainder quotient modulo remainder gcd lcm numerator denominator floor ceiling truncate round rationalize square exact-integer-sqrt expt inexact exact number->string string->number

Booleans
not boolean? boolean=?

Pairs and lists
pair? cons car cdr set-car! set-cdr! caar cadr cdar cddr null? list?
make-list list length append reverse list-tail list-ref list-set!
memq memv member assq assv assoc list-copy

Symbols
symbol? symbol=? symbol->string string->symbol

Characters

Strings
string? make-string string string-length string-ref string-set!
string=? string<? string>? string<=? string>=? substring string-append
string->list list->string string-copy string-copy! string-fill!
string-map string-for-each
Chapter 10: Library modules - R7RS standard libraries

Vectors

- `vector?`
- `make-vector`
- `vector-length`
- `vector-ref`
- `vector-set!`
- `vector->list`
- `list->vector`
- `vector->string`
- `string->vector`
- `vector-copy`
- `vector-copy!`
- `vector-append`
- `vector-fill!`

Control features

- `procedure?`
- `apply`
- `map`
- `call-with-current-continuation`
- `call/cc`
- `values`
- `call-with-values`
- `dynamic-wind`

Exception

- `error`

Environments and evaluation

- `scheme-report-environment`
- `null-environment`

Input and output

- `input-port?`
- `output-port?`
- `port?`
- `current-input-port`
- `current-output-port`
- `current-error-port`
- `close-port`
- `close-input-port`
- `close-output-port`
- `open-input-string`
- `open-output-string`
- `get-output-string`
- `read-char`
- `peek-char`
- `read-u8`
- `peek-u8`
- `read-line`
- `eof-object?`
- `eof-object?`
- `char-ready?`
- `u8-ready?`
- `newline`
- `write-char`
- `write-u8`

Bytevector utilities

R7RS's bytevectors are the same as Gauche's u8vectors.

The following procedures are the same as `gauche.uvector`'s (see Section 9.38.5 [Bytevector compatibility], page 569).

- `bytevector`
- `bytevector?`
- `make-bytevector`
- `bytevector-length`
- `bytevector-u8-ref`
- `bytevector-u8-set!`
- `bytevector-copy`
- `bytevector-copy!`
- `bytevector-append`
- `read-bytevector`
- `read-bytevector!`
- `write-bytevector`

The following procedures are the same as `gauche.vport`'s (see Section 9.40 [Virtual ports], page 572).

- `open-input-bytevector`
- `open-output-bytevector`
- `get-output-bytevector`

And the following procedures are the same as `gauche.unicode`'s (see Section 9.37.1 [Unicode transfer encodings], page 550).

- `utf8->string`
- `string->utf8`

Control features

The procedure `with-exception-handler` is the same as Gauche's built-in. See Section 6.19.3.3 [Low-level exception handling mechanism], page 252, for the explanation.

- `raise obj`  [Function]
- `raise-continuable obj`  [Function]

[R7RS base] {scheme.base} Gauche's `raise` may return if `obj` isn't a `<serious-condition>`. Distinguishing continuable and noncontinuable exception throw by the procedure has an issue when your exception handler wants to reraise the condition (you don’t know if the original condition is raised by `raise` or `raise-continuable`!). Yet R7RS adopted that model, so we compel.

R7RS `raise` is a wrapper of Gauche’s `raise`, which throws an error if Gauche’s `raise` returns. R7RS `raise-continuable` is currently just an alias of Gauche’s `raise`—as long as you don’t pass `<serious-condition>`, it may return. It is not exactly R7RS conformant—it won’t
return if you pass `<serious-condition>` or object of one of its subclasses (e.g. `<error>`), but it’s weird to expect returning from raising `<error>`, isn’t it?

```scheme
(error-object? exc) [Function]
[R7RS base] {scheme.base} Defined as (condition-has-type? exc <error>))
```

```scheme
(error-object-message exc) [Function]
[R7RS base] {scheme.base} If exc is a `<message-condition>`, returns its message-prefix slot; otherwise, returns an empty string.
```

```scheme
(error-object-irritants exc) [Function]
[R7RS base] {scheme.base} If exc is a `<message-condition>`, returns its message-args slot; otherwise, returns an empty string.
```

```scheme
(read-error? exc) [Function]
[R7RS base] {scheme.base} Defined as (condition-has-type? e <read-error>)).
```

```scheme
(file-error? exc) [Function]
[R7RS base] {scheme.base} At this moment, Gauche doesn’t have distinct `<file-error>` condition, but most file errors are thrown as one of `<system-error>`s. This procedure checks error code of `<system-error>` and returns #t if the error is likely to be related to the filesystem.
```

**Input and output**

```scheme
(textual-port? port) [Function]
```

```scheme
(binary-port? port) [Function]
```

```scheme
(input-port-open? iport) [Function]
```

```scheme
(output-port-open? oport) [Function]
```

```scheme
(flush-output-port :optional oport) [Function]
```

```scheme
(features) [Function]
```

10.2.3 scheme.case-lambda - R7RS case-lambda

**scheme.case-lambda** [Module]

Exports bindings of R7RS (scheme case-lambda) library. From R7RS programs, those bindings are available by `(import (scheme case-lambda))`.

The only binding exported from this module is case-lambda, and it is the same as Gauche’s built-in case-lambda; see Section 4.3 [Making procedures], page 47, for the details.

10.2.4 scheme.char - R7RS char library

**scheme.char** [Module]

Exports bindings of R7RS (scheme char) library. From R7RS programs, those bindings are available by `(import (scheme char))`. 

Chapter 10: Library modules - R7RS standard libraries

The following procedures are the same as Gauche's builtin procedures; see Section 6.9 [Characters], page 160.

- char-alphabetic?
- char-ci<=?
- char-ci<?
- char-ci=?
- char-ci=>?
- char-downcase
- char-foldcase
- char-lower-case?
- char-numeric?
- char-upcase
- char-upper-case?
- char-whitespace?

The following procedures are the same as the ones provided in `gauche.unicode` module (see Section 9.37.3 [Full string case conversion], page 554). They use full case folding by Unicode standard (e.g. taking into account of German eszett).

- string-ci<=?
- string-ci<?
- string-ci=?
- string-ci=>?
- string-ci>?
- string-downcase
- string-foldcase
- string-upcase

**digit-value**

*Function*

If `c` is a character with Nd general category—that is, if it represents a decimal digit—this procedure returns the value the character represents. Otherwise it returns `#f`.

- `(digit-value #\3)` ⇒ 3
- `(digit-value #\z)` ⇒ `#f`

Note that Unicode defines about two dozen sets of digit characters.

- `(digit-value #\x11068)` ⇒ 2

Gauche's built-in procedure `digit->integer` has more general interface (see Section 6.9 [Characters], page 160).

- `(digit-value c) ≡ (digit->integer c 10 #t)`

### 10.2.5 scheme.complex - R7RS complex numbers

**scheme.complex**

*Module*

Exports bindings of R7RS (scheme complex) library. From R7RS programs, those bindings are available by `(import (scheme complex))`.

This module provides the following bindings, all of which are Gauche built-in (see Section 6.3.5 [Numerical conversions], page 134).

- angle
- imag-part
- magnitude
- make-polar
- make-rectangular
- real-part

### 10.2.6 scheme.cxr - R7RS cxr accessors

**scheme.cxr**

*Module*

Exports bindings of R7RS (scheme cxr) library. From R7RS programs, those bindings are available by `(import (scheme cxr))`.

This module provides the following bindings, all of which are Gauche built-in (see Section 6.6.5 [List accessors and modifiers], page 143).

- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
- caaar
- caadr
- cadar
- caddr
Chapter 10: Library modules - R7RS standard libraries

**eval expr environment**

[R7RS eval] {scheme.eval} This is the same as Gauche’s built-in eval (see Section 6.20 [Eval and repl], page 258).

**environment import-list ...**

[R7RS eval] {scheme.eval} This is R7RS way to create an environment specifier suitable to pass to eval. In Gauche, an environment specifier is just a module object.

The argument is the same as what r7rs#import takes. This procedure creates an empty environment (as a fresh anonymous module; see make-module in Section 4.13.6 [Module introspection], page 84, for the details), then imports the bindings as specified by import-lists.

The following example creates an environment that includes scheme.base bindings plus select-module syntax from Gauche.

```
(environment
  '(scheme base)
  '(only (gauche base) select-module))
⇒ #<module #f> ; an anonymous module
```

### 10.2.8 scheme.file - R7RS file library

**scheme.file**

[Module]

Exports bindings of R7RS (scheme file) library. From R7RS programs, those bindings are available by (import (scheme file)).

The following bindings provided in this module are Gauche built-in (see Section 6.21.4 [File ports], page 264, and Section 6.24.4.4 [File stats], page 301).

- call-with-input-file
- call-with-output-file
- file-exists?
- open-input-file
- open-output-file
- with-input-from-file
- with-output-to-file

The following binding is the same as one in file.util (see Section 12.34.4 [File operations], page 888).

- delete-file

**open-binary-input-file filename**

[R7RS file] {scheme.file} In Gauche, ports are both textual and binary at the same time, so these R7RS procedures are just aliases of open-input-file and open-output-file, respectively. See Section 6.21.4 [File ports], page 264.

### 10.2.9 scheme.inexact - R7RS inexact numbers

**scheme.inexact**

[Module]

Exports bindings of R7RS (scheme inexact) library. From R7RS programs, those bindings are available by (import (scheme inexact)).

This module provides the following bindings, all of which are Gauche built-in (see Section 6.3.4 [Arithmetics], page 126, and Section 6.3.2 [Numerical predicates], page 123).

- acos
- asin
- atan
- cos
- exp
- finite?
- infinite?
- log
- nan?
- sin
- sqrt
- tan
10.2.10 scheme.lazy - R7RS lazy evaluation

scheme.lazy

Exports bindings of R7RS (scheme lazy) library. From R7RS programs, those bindings are available by (import (scheme lazy)).

The following bindings this module provides are Gauche built-ins (see Section 6.18.1 [Delay force and lazy], page 239).

delay force promise?
delay-force promise

[R7RS lazy] {scheme.lazy} This is the same as Gauche’s built-in lazy. see Section 6.18.1 [Delay force and lazy], page 239, for the discussion about when this form should be used.

make-promise obj

[R7RS lazy] {scheme.lazy} If obj is a promise, it is returned as is. Otherwise, A promise, which yields obj when forced, is returned. Because this is a procedure, expression passed as obj is eagerly evaluated, so this doesn’t have effect on lazy evaluation, but can be used to ensure you have a promise.

This procedure is important on implementations where force only takes a promise, and portable code should use this procedure to yield a value that can be passed to force.

If you write Gauche-specific code, however, force can take non-promise values, so you don’t need this.

Note: Gauche has built-in make-promise, which behaves as defined in SRFI-226 and slightly different from this one (see Section 6.18.1 [Delay force and lazy], page 239).

10.2.11 scheme.load - R7RS load

scheme.load

Exports bindings of R7RS (scheme load) library. From R7RS programs, those bindings are available by (import (scheme load)).

load file :optional env

[R7RS load] {scheme.load} R7RS load takes environment as an optional argument, while Gauche load takes it as a keyword argument (among other keyword arguments). See Section 6.22.1 [Loading Scheme file], page 286.

In Gauche, env is just a module. In portable code, you can create a module with desired bindings with R7RS environment procedure; see Section 10.2.7 [R7RS eval], page 588.

10.2.12 scheme.process-context - R7RS process context

scheme.process-context

Exports bindings of R7RS (scheme process-context) library. From R7RS programs, those bindings are available by (import (scheme process-context)).

The following bindings are the same as Gauche built-ins (see Section 6.24.2 [Command-line arguments], page 294, and Section 6.24.1 [Program termination], page 293):

command-line exit

The following bindings are the same as SRFI-98 (see Section 11.20 [Accessing environment variables], page 727):

get-environment-variable get-environment-variables
emergency-exit :optional (obj 0)  [Function]
[R7RS process-context] {scheme.process-context} Terminate the program without running any clean-up procedures (after thunks of dynamic-wind). I/O buffers won’t be flushed. Internally, it calls the _exit(2) system call directly. The optional argument is used for the process exit code. When omitted, 0 is assumed.
This is almost the same as Gauche’s sys-exit, except that sys-exit requires the exit code argument (see Section 6.24.1 [Program termination], page 293).

10.2.13 scheme.read - R7RS read

scheme.read  [Module]
Exports bindings of R7RS (scheme read) library. From R7RS programs, those bindings are available by (import (scheme read)).
The only binding exported from this module is read, which is the same as Gauche’s built-in. See Section 6.21.7.1 [Reading data], page 270.

10.2.14 scheme.repl - R7RS repl

scheme.repl  [Module]
Exports bindings of R7RS (scheme repl) library. From R7RS programs, those bindings are available by (import (scheme repl)).
The only binding exported from this module is interaction-environment, which is the same as Gauche’s built-in. See Section 6.20 [Eval and repl], page 258.

10.2.15 scheme.time - R7RS time

scheme.time  [Module]
Exports bindings of R7RS (scheme time) library. From R7RS programs, those bindings are available by (import (scheme time)).
current-second  [Function]
[R7RS time] {scheme.time} Returns a real number represents the number of seconds since the midnight of Jan. 1, 1970 TAI (which is 23:59:52, Dec 31, 1969 UTC, that is, -8 seconds before Unix Epoch.) Number of leap seconds were inserted since then, and as of 2017, UTC is 37 seconds behind TAI. That means the number returned is 29 seconds larger than the unix time, which is returned from sys-time or sys-gettimeofday.
The reason that R7RS adopts TAI is that it is monotonic and suitable to take difference of two timepoints. The unix time returned by sys-time and sys-gettimeofday are defined in terms of UTC date and time, so if the interval spans across leap seconds, it won’t reflect the actual number of seconds in the interval. (The precise definition is given in section 4.15 of IEEE Std 1003.1, 2013 Edition, a.k.a Single Unix Specification 4.)
However, since we don’t know yet when the next leap second happen, the current implementation just uses a fixed amount of offset from the unix time.
Just be aware the difference, or you’ll be surprised if you pass the return value of current-second to the UTC time formatter such as sys-strftime, or compare it with the file timestamps which uses the unix time. You can convert between TAI and UTC using SRFI-19 (see Section 11.7.4 [SRFI-19 Date], page 703).
current-jiffy  [Function]
[R7RS time] {scheme.time} Returns an exact integer measuring a real (wallclock) time elapsed since some point in the past, which does not change while a process is running. The time unit is a jiffies-per-second-th second.
The absolute value of current jiffies doesn’t matter, but the difference can be used to measure the time interval.
**jiffies-per-second**

**[Function]**

**R7RS time** {scheme.time} Returns a constant to tell how many time units used in current-jiffy consists of a second. In the current Gauche implementation, this is $10^9$ on 64bit architectures (that is, nanosecond resolution) and $10^4$ on 32bit architectures (100 microseconds resolution).

The resolution for 32bit architectures is unfortunately rather coarse, but if we make it finer the current jiffy value easily becomes bignums, taking time to allocate and operate, beating the purpose of benchmarking. With the current choice, we have 53,867 seconds since process start before we spill into bignum on 32bit architecture. On 64bit architectures we have enough bits not to worry about bignums, with nanosecond resolution.

If you want to do more finer benchmarks on 32bit machines, you need to roll your own with sys-clock-gettime-monotonic or sys-gettimeofday.

### 10.2.16 scheme.write - R7RS write

**scheme.write**

**[Module]**

Exports bindings of R7RS (scheme write) library. From R7RS programs, those bindings are available by (import (scheme write)).

This module provides the following bindings, all of which are Gauche built-in (see Section 6.21.8.3 [Object output], page 277).

- display
- write
- write-shared
- write-simple

### 10.2.17 scheme.r5rs - R5RS compatibility

**scheme.r5rs**

**[Module]**

This module is to provide R5RS environment in R7RS programs. The following bindings are exported. Note that lambda is scheme#lambda, without the support of extended formals (:optional etc.) See Section 4.3 [Making procedures], page 47, for the details of extended formals.

* + - / < <= = > >= abs acos and angle append apply asin assoc assq assv atan begin boolean? caaar caaadr caaar caaddr caadr caaddr
Chapter 10: Library modules - R7RS standard libraries

sqrt string string->list string->number string->symbol string-append
string-copy string-fill! string-length string-ref string-set!
symbol->string symbol? tan truncate values vector vector->list
vector-fill! vector-length vector-ref vector-set! vector?
with-input-from-file with-output-to-file write write-char zero?

10.3 R7RS large

R7RS large is still under development, and we’re gradually adding support of the libraries that has been passed.

Currently R7RS-large has two editions (Red and Tangerine), and Gauche supports them.

10.3.1 scheme.list - R7RS lists

scheme.list [Module]

This module is a rich collection of list manipulation procedures, and same as srfi.1.

Note that Gauche supports quite a few scheme.list procedures as built-in. The following procedures can be used without loading scheme.list module. For the manual entries of these procedures, Section 6.6 [Pairs and lists], page 140.

null-list? cons* last member
take drop take-right drop-right take! drop-right!
delete delete! delete-duplicates delete-duplicates!
assoc alist-copy alist-delete alist-delete!
any every filter filter! fold fold-right find find-tail
split-at split-at! iota

List constructors

xcons cd ca [Function]

[R7RS list] {scheme.list} Equivalent to (cons ca cd). Useful to pass to higher-order procedures.

list-tabulate n init-proc [Function]

[R7RS list] {scheme.list} Constructs an n-element list, in which each element is generated by (init-proc i).

(list-tabulate 4 values) ⇒ (0 1 2 3)

circular-list elt1 elt2 ... [Function]

[R7RS list] {scheme.list} Constructs a circular list of the elements.

(circular-list 'z 'q) ⇒ (z q z q z q ...)

List predicates

not-pair? x [Function]

[R7RS list] {scheme.list} Same as (lambda (x) (not (pair? x))).

SRFI-1 says: Provided as a procedure as it can be useful as the termination condition for list-processing procedures that wish to handle all finite lists, both proper and dotted.

list= elt= list ... [Function]

[R7RS list] {scheme.list} Determines list equality by comparing every n-th element of given lists by the procedure elt=.
It is an error to apply list= to anything except proper lists.

The equality procedure must be consistent with eq?, i.e.

\[(\text{eq? } x y) \Rightarrow (\text{elt=} x y)\].

**List selectors**

- **first** [Function]
- **second** [Function]
- **third** [Function]
- **fourth** [Function]
- **fifth** [Function]
- **sixth** [Function]
- **seventh** [Function]
- **eighth** [Function]
- **ninth** [Function]
- **tenth** [Function]

[R7RS list] \{\text{scheme.list}\} Returns n-th element of the (maybe improper) list.

- **car+cdr** [Function]

[R7RS list] \{\text{scheme.list}\} Returns two values, (car pair) and (cdr pair).

**List miscellaneous routines**

- **zip** [Function]

[R7RS list] \{\text{scheme.list}\} Equivalent to (map list clist1 clist2 ...). If zip is passed n lists, it returns a list as long as the shortest of these lists, each element of which is an n-element list comprised of the corresponding elements from the parameter lists.

\[
(\text{zip } '(\text{one two three}) \\
\quad '(1 2 3) \\
\quad '(\text{odd even odd even odd even odd even})) \\
\Rightarrow ((1 \text{ odd}) (2 \text{ even}) (3 \text{ odd}))
\]

\[
\text{At least one of the argument lists must be finite:}
\]

\[
(\text{zip } '(3 1 4 1) (\text{circular-list #f #t})) \\
\Rightarrow ((3 \text{ #f}) (1 \text{ #t}) (4 \text{ #f}) (1 \text{ #t}))
\]

- **unzip1** [Function]
- **unzip2** [Function]
- **unzip3** [Function]
- **unzip4** [Function]
- **unzip5** [Function]

[R7RS list] \{\text{scheme.list}\} unzip1 takes a list of lists, where every list must contain at least one element, and returns a list containing the initial element of each such list. unzip2 takes a list of lists, where every list must contain at least two elements, and returns two values: a list of the first elements, and a list of the second elements. unzip3 does the same for the first three elements of the lists, and so on.

\[
(\text{unzip2 } '((1 \text{ one}) (2 \text{ two}) (3 \text{ three})) \Rightarrow \\
\quad (1 2 3) \text{ and} \\
\quad (\text{one two three})
\]
List fold, unfold & map

\textbf{pair-fold} \textit{kons knil c1ist1 c1ist2} . . . \quad \textbf{[Function]}

\textbf{pair-fold-right} \textit{kons knil c1ist1 c1ist2} . . . \quad \textbf{[Function]}

$\{\text{R7RS list}\} \{\text{scheme.list}\}$ Like \textit{fold} and \textit{fold-right}, but the procedure \textit{kons} gets each \textit{cdr} of the given \textit{clists}, instead of \textit{car}.

\[(\text{pair-fold cons '()} '(a b c d e)) \Rightarrow ((e) (d e) (c d e) (b c d e) (a b c d e))\]

\[(\text{pair-fold-right cons '()} '(a b c d e)) \Rightarrow ((a b c d e) (b c d e) (c d e) (d e) (e))\]

\textbf{unfold} \textit{p f g seed} :optional \textit{tail-gen} \quad \textbf{[Function]}

$\{\text{R7RS list}\} \{\text{scheme.list}\}$ Fundamental recursive list constructor. Defined by the following recursion.

\[(\text{unfold p f g seed tail-gen}) \equiv \]
\[
  \begin{cases}
  \text{if} (\text{p seed}) \\
  \quad \text{(tail-gen seed)} \\
  \quad \quad \text{(cons (f seed)} \\
  \quad \quad \quad \text{(unfold p f g (g seed))})
  \end{cases}
\]

That is, \textit{p} determines where to stop, \textit{g} is used to generate successive seed value from the current seed value, and \textit{f} is used to map each seed value to a list element.

\[(\text{unfold (pa$ = 53$) integer->char (pa$ + 1$) 48}) \Rightarrow (\#\text{\textbackslash 0} \#\text{\textbackslash 1} \#\text{\textbackslash 2} \#\text{\textbackslash 3} \#\text{\textbackslash 4})\]

\textbf{unfold-right} \textit{p f g seed} :optional \textit{tail} \quad \textbf{[Function]}

$\{\text{R7RS list}\} \{\text{scheme.list}\}$ Fundamental iterative list constructor. Defined by the following recursion.

\[(\text{unfold-right p f g seed tail}) \equiv \]
\[
  \begin{cases}
  \text{let lp ((seed seed) (lis tail))} \\
  \quad (\text{if} (\text{p seed}) \\
  \quad \quad \text{lisp} \\
  \quad \quad \quad \text{(lp (g seed) (cons (f seed) lisp)))})
  \end{cases}
\]

\[(\text{unfold-right (pa$ = 53$) integer->char (pa$ + 1$) 48}) \Rightarrow (\#\text{\textbackslash 4} \#\text{\textbackslash 3} \#\text{\textbackslash 2} \#\text{\textbackslash 1} \#\text{\textbackslash 0})\]

\textbf{map!} \textit{f c1ist1 c1ist2} . . . \quad \textbf{[Function]}

$\{\text{R7RS list}\} \{\text{scheme.list}\}$ The procedure \textit{f} is applied to each element of \textit{c1ist1} and corresponding elements of \textit{c1ist2s}, and the result is collected to a list. Cells in \textit{c1ist1} is reused to construct the result list.

\textbf{map-in-order} \textit{f c1ist1 c1ist2} . . . \quad \textbf{[Function]}

$\{\text{R7RS list}\} \{\text{scheme.list}\}$ A variant of \textit{map}, but it guarantees to apply \textit{f} on each elements of arguments in a left-to-right order. Since Gauche’s \textit{map} implementation follows the same order, this function is just a synonym of \textit{map}.

\textbf{pair-for-each} \textit{f c1ist1 c1ist2} . . . \quad \textbf{[Function]}

$\{\text{R7RS list}\} \{\text{scheme.list}\}$ Like \textit{for-each}, but the procedure \textit{f} is applied on \textit{clists} themselves first, then each \textit{cdrs} of them, and so on.

\[(\text{pair-for-each write '}(a b c)) \Rightarrow \text{prints} (a \ b \ c)(b \ c)(c)\]
List partitioning

\texttt{partition pred list}  \hfill [Function]  
\texttt{partition! pred list}  \hfill [Function]  

[R7RS list] \{scheme.list\} filter and remove simultaneously, i.e. returns two lists, the first is the result of filtering elements of list by \texttt{pred}, and the second is the result of removing elements of list by \texttt{pred}.

\begin{verbatim}
(partition odd? '(3 1 4 5 9 2 6))
⇒ (3 1 5 9) (4 2 6)
\end{verbatim}

\texttt{partition!} is the linear-update variant. It may destructively modifies list to produce the result.

List searching

\texttt{take-while pred clist}  \hfill [Function]  
\texttt{take-while! pred list}  \hfill [Function]  

[R7RS list] \{scheme.list\} Returns the longest initial prefix of clist whose elements all satisfy \texttt{pred}.

\texttt{drop-while pred clist}  \hfill [Function]  

[R7RS list] \{scheme.list\} Drops the longest initial prefix of clist whose elements all satisfy \texttt{pred}, and returns the rest.

\texttt{span pred clist}  \hfill [Function]  
\texttt{span! pred list}  \hfill [Function]  
\texttt{break pred clist}  \hfill [Function]  
\texttt{break! pred list}  \hfill [Function]  

[R7RS list] \{scheme.list\} \texttt{span} is equivalent to (\texttt{values (take-while pred clist) (drop-while pred clist)}). \texttt{break} inverts the sense of \texttt{pred}.

\texttt{list-index pred clist1 clist2 \ldots}  \hfill [Function]  

[R7RS list] \{scheme.list\} Returns the index of the leftmost element that satisfies \texttt{pred}. If no element satisfies \texttt{pred}, \texttt{#f} is returned.

Association lists

\texttt{alist-cons key datum alist}  \hfill [Function]  

[R7RS list] \{scheme.list\} Returns (\texttt{cons (cons key datum) alist}). This is an alias of the Gauche builtin procedure \texttt{acons}.

Lists as sets

These procedures use a list as a set, that is, the elements in a list matter, but their order doesn’t.

All procedures in this category takes a comparison procedure \texttt{elt=}, as the first argument, which is used to determine two elements in the given sets are the same.

Since lists require linear time to search, those procedures aren’t suitable to deal with large sets. See Section 10.3.5 [R7RS sets], page 606, if you know your sets will contain more than a dozen items or so.

See also Section 12.81 [Combination library], page 1015, which concerns combinations of elements in the set.

\texttt{lset<= elt= list1 \ldots}  \hfill [Function]  

[R7RS list] \{scheme.list\} Returns \texttt{#t} iff all elements in list1 are also included in list2, and so on. If no lists are given, or a single list is given, \texttt{#t} is returned.
\begin{itemize}
\item \textbf{lset=} \textit{elt=} \textit{list1 list2} \ldots
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Returns \#t} if all elements in \textit{list1} are in \textit{list2}, and all elements in \textit{list2} are in \textit{list1}, and so on.

\[(\text{lset=} \text{eq?} \ [('b\ e\ a)\ ('a\ e\ b)\ ('e\ e\ b\ a)] \Rightarrow \#t)\]

\begin{itemize}
\item \textbf{lset-adjjoin \textit{elt=} \textit{list} \textit{elt} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Adds \textit{elt} \ldots} to the set \textit{list}, if each one is not already a member of \textit{list}. (The order doesn’t matter).

\[(\text{lset-adjjoin eq?} \ '(a\ b\ c)\ '(a\ e) \Rightarrow '\(a\ e\ b\ c)\]

\begin{itemize}
\item \textbf{lset-union \textit{elt=} \textit{list1} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Returns the union of the sets \textit{list1} \ldots.}

\begin{itemize}
\item \textbf{lset-intersection \textit{elt=} \textit{list1 list2} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Returns a set of elements that are in every \textit{lists}.}

\begin{itemize}
\item \textbf{lset-difference \textit{elt=} \textit{list1 list2} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Returns a set of elements that are in \textit{list1} but not in \textit{list2}. In n-ary case, binary difference operation is simply folded.}

\begin{itemize}
\item \textbf{lset-xor \textit{elt=} \textit{list1} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Returns the exclusive-or of given sets; that is, the returned set consists of the elements that are in either \textit{list1} or \textit{list2}, but not in both. In n-ary case, binary xor operation is simply folded.}

\begin{itemize}
\item \textbf{lset-diff+intersection \textit{elt=} \textit{list1 list2} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Returns two sets, a difference and an intersection of given sets.}

\begin{itemize}
\item \textbf{lset-union! \textit{elt=} \textit{list} \ldots}
\item \textbf{lset-intersection! \textit{elt=} \textit{list1 list2} \ldots}
\item \textbf{lset-difference! \textit{elt=} \textit{list1 list2} \ldots}
\item \textbf{lset-xor! \textit{elt=} \textit{list1} \ldots}
\item \textbf{lset-diff+intersection! \textit{elt=} \textit{list1 list2} \ldots}
\end{itemize}

\textbf{[Function]} \{\textit{scheme.list}\} \textbf{Linear update variant of the corresponding procedures. The cells in the first list argument may be reused to construct the result.}

\subsection*{10.3.2 \textit{scheme.vector} - R7RS vectors}

\textbf{\textit{scheme.vector}} \{\textit{module}\}

\textbf{This module adds rich set of vector operations to the built-in / R7RS vector procedures.}

\textbf{The following procedures are built-in. See Section 6.13.1 \{Vectors\}, page 195, for the description. We only explain the procedures that are not built-in.}

\begin{itemize}
\item \textbf{make-vector} \rightarrow \textit{vector} \rightarrow \textit{vector?}
\item \textbf{vector-ref} \rightarrow \textit{vector-set!} \rightarrow \textit{vector-length}
\item \textbf{vector-fill!} \rightarrow \textit{vector-copy} \rightarrow \textit{vector-copy!}
\item \textbf{vector-append} \rightarrow \textit{vector->list} \rightarrow \textit{list->vector}
\item \textbf{reverse-list->vector} \rightarrow \textit{vector->string} \rightarrow \textit{string->vector}
\item \textbf{vector-map} \rightarrow \textit{vector-map!} \rightarrow \textit{vector-for-each}
\end{itemize}

\textbf{This module is \textit{srfi.133}, which supersedes \textit{srfi.43} (see Section 11.12 \{Vector library (Legacy)\}, page 717). Note that the interface of following procedures in \textit{srfi.43} are changed for the consistency:}

\begin{itemize}
\item \textbf{vector-map} \rightarrow \textit{vector-map!} \rightarrow \textit{vector-for-each}
\item \textbf{vector-fold} \rightarrow \textit{vector-fold-right} \rightarrow \textit{vector-count}
\end{itemize}
Some of the functionalities of srfi.43 version is supported by built-in procedures (e.g. Built-in vector-map-with-index is the same as srfi.43's vector-map). So there's little point for new code to use srfi.43.

Vector constructors

vector-unfold \( f \) length seed ...  
[Function]  
[R7RS vector] {scheme.vector} Creates a vector of length length, filling elements left to right by calling \( f \) repeatedly.

The procedure \( f \) must take as many arguments as one plus number of seed values, and must return the same number of values. The first argument is the index. The first return value is used for the element of the result vector, and the rest of return values are passed to the next call of \( f \).

\[
\begin{align*}
\text{(vector-unfold (^[i] (* i i)) 5)} & \Rightarrow \#(0 1 4 9 16) \\
\text{(vector-unfold (^[i x] (values (cons i x) (* x 2))) 8 1)} & \Rightarrow \#((0 . 1) (1 . 2) (2 . 4) (3 . 8) \\
& \quad (4 . 16) (5 . 32) (6 . 64) (7 . 128))
\end{align*}
\]

Note: This protocol is different from the list unfold and most of other \*-unfold constructors (see Section 10.3.1 [R7RS lists], page 593). For fixed-length structures like vectors, it makes more sense to know the final length beforehand, rather than checking the termination condition for each iteration.

vector-unfold-right \( f \) length seed ...  
[Function]  
[R7RS vector] {scheme.vector} Creates a vector of length length, filling elements right to left by calling \( f \) repeatedly.

The procedure \( f \) must take as many arguments as one plus number of seed values, and must return the same number of values. The first argument is the index. The first return value is used for the element of the result vector, and the rest of return values are passed to the next call of \( f \).

\[
\begin{align*}
\text{(vector-unfold-right (^[i] (* i i)) 5)} & \Rightarrow \#(0 1 4 9 16) \\
\text{(vector-unfold-right (^[i x] (values (cons i x) (* x 2))) 8 1)} & \Rightarrow \#((0 . 128) (1 . 64) (2 . 32) (3 . 16) \\
& \quad (4 . 8) (5 . 4) (6 . 2) (7 . 1))
\end{align*}
\]

vector-reverse-copy vec :optional start end  
[Function]  
[R7RS vector] {scheme.vector} Copies the vector vec with reversing its elements. Optional start and end arguments can limit the range of the input.

\[
\begin{align*}
\text{(vector-reverse-copy '(#(a b c d e) 1 4))} & \Rightarrow \#(d c b)
\end{align*}
\]

vector-concatenate list-of-vectors  
[Function]  
[R7RS vector] {scheme.vector} Same as (apply vector-append list-of-vectors).

vector-append-subvectors spec ...  
[Function]  
[R7RS vector] {scheme.vector} The number of arguments must be multiple of 3. The argument list must be in the following format, where each vecN is a vector, and startN and endN are nonnegative exact integers:

\[
\begin{align*}
\text{vec1 start1 end1 vec2 start2 end2 ...}
\end{align*}
\]
This procedure creates a new vector by concatenating subvectors specified by each triplet. That is, it works as if it’s the following code, except it avoids copying each subvector:

\[
\text{(vector-append (vector-copy vec1 start1 end1) (vector-copy vec2 start2 end2) \ldots)}
\]

Here’s an example:

\[
\text{(vector-append-subvectors '(#(a b c d e) 0 3 '(#f g h i j) 2 5)}
\]

⇒ #(#a b c h i j)

Vector predicates

\text{vector-empty? vec} \quad [\text{Function}]
\text{vector-empty?} \quad \text{vec} \quad \text{(R7RS vector)} \quad \text{\{scheme.vector\}} \quad \text{Returns \#t if \text{vec’s} length is zero, and \#f if \text{vec’s} length is more than zero. Signals an error if \text{vec} is not a vector.}

\text{vector=? elt vec \ldots} \quad [\text{Function}]
\text{vector=} \quad \text{elt} \quad \text{vec} \quad \text{(R7RS vector)} \quad \text{\{scheme.vector\}} \quad \text{Compares vecs element-wise, using given predicate elt=}. \text{Returns \#t iff lengths of all the vectors are the same, and every corresponding elements are equal by elt=}. \text{Elt=} \quad \text{is always called with two arguments and must return \#t iff two are the same.}
\text{When zero or one vector is passed, elt=} \quad \text{won’t be called and vector=} \quad \text{always returns \#t.}

Vector iteration

\text{vector-fold kons knil vec1 vec2 \ldots} \quad [\text{Function}]
\text{vector-fold} \quad \text{kons} \quad \text{knil} \quad \text{vec1} \quad \text{vec2} \quad \text{(R7RS vector)} \quad \text{\{scheme.vector\}} \quad \text{Kons is a procedure that takes \(n+1\) arguments, where \(n\) is the number of given vectors. For each element of the given vectors, \text{kons} is called as (kons seed e_{1i} e_{2i} \ldots), where and e_{ni} is the \(i\)-th element of the vector \(n\). If the lengths of the vectors differ, iteration stops when the shortest vector is exhausted.}
\text{The initial value of seed is \text{knil}, and the return value from \text{kons} is used as the next seed value.}
\text{The last return value of kons is returned from vector-fold.}
\text{The iteration is strictly left to right.}
\text{Note that the seed value precedes elements, which is opposite to fold (see Section 9.6.1 [Mapping over collection], page 405). It’s an unfortunate historical glitch; vector-fold-left would be more consistent name.}

\[
\text{(vector-fold (\^[a b] (cons b a)) \^() \'#(a b c d))}
\]

⇒ (d c b a)

\text{vector-fold-right kons knil vec1 vec2 \ldots} \quad [\text{Function}]
\text{vector-fold-right} \quad \text{kons} \quad \text{knil} \quad \text{vec1} \quad \text{vec2} \quad \text{(R7RS vector)} \quad \text{\{scheme.vector\}} \quad \text{Like vector-fold, but elements in the vec1 vec2 \ldots are visited from right to left.}
\text{If the lengths of the vectors differ, this procedure only looks up to the number of elements of the shortest vector and ignores any excessive elements. See the second example below.}
\text{Unlike fold-right (see Section 9.31.3 [Mapping over sequences], page 511), the procedure kons takes the seed value in the first argument.}

\[
\text{(vector-fold-right (\^[a b] (cons b a)) \^() \'#(a b c d))}
\]

⇒ (a b c d)

\[
\text{(vector-fold-right (\^[s x y] (cons (list x y) s)) \^() \'#(a b c) \'#(1 2 3 4))}
\]

⇒ ((a 1) (b 2) (c 3))
vector-count \( \text{pred vec1 vec2} \ldots \) 
[Function]  
[R7RS vector] \{scheme.vector\} Applies \text{pred} on each element in argument vectors (if \( N \) vectors are given, \text{pred} takes \( N \) arguments, the first being \( i \)-th element of \text{vec1}, the second being \( i \)-th element of \text{vec2}, etc.) Then returns the number of times \text{pred} returned true value. The order \text{pred} applied to each element is unspecified.  
\[(\text{vector-count odd? '}(0 \ 1 \ 2 \ 3 \ 4)) \Rightarrow 2\]
\[(\text{vector-count < '}(7 \ 3 \ 9 \ 1 \ 5) '}(6 \ 8 \ 2 \ 3 \ 8 \ 8)) \Rightarrow 3\]

vector-cumulate \( f \text{ seed vec} \) 
[Function]  
[R7RS vector] \{scheme.vector\} Returns a fresh vector with the same size of \text{vec}, with the elements calculated as follows:  
The first element of result vector is a result of procedure \( f \) called with \text{seed} and the first element of \text{vec}.  
The \( i \)-th element of result vector is a result of procedure \( f \) called with \( i-1 \)-th element of result vector and \( i \)-th element of \text{vec}.  
\[(\text{vector-cumulate string-append "z" '}("a" "b" "c")) \Rightarrow '("za" "zab" "zabc")\]

Vector searching

vector-index \( \text{pred vec1 vec2} \ldots \) 
[Function]  
vector-index-right \( \text{pred vec1 vec2} \ldots \) 
[Function]  
[R7RS vector] \{scheme.vector\} Returns the index of the first or the last elements in \text{vec1 vec2} \ldots that satisfy \text{pred}, respectively. Returns \#f if no elements satisfy \text{pred}. In vector-index, comparison ends at the end of the shortest vector. For vector-index-right, all the vectors must have the same length.

vector-skip \( \text{pred vec1 vec2} \ldots \) 
[Function]  
vector-skip-right \( \text{pred vec1 vec2} \ldots \) 
[Function]  
[R7RS vector] \{scheme.vector\} Like vector-index and vector-index-right, except that the result of \text{pred} is negated. That is, returns the index of the first or the last elements that don’t satisfy \text{pred}.

vector-binary-search \( \text{vec value cmp} \)  
[Optional] \{scheme.vector\} Look for \text{value} in a sorted vector \text{vec}, and returns its index if it is found, or \#f if it is not found.  
Comparison of \text{value} and an element in \text{vec} is done by a procedure \text{cmp}, which takes two arguments, and should return a negative integer if the first argument is less than the second, 0 if they are the same, and a positive integer if the first is greater than the second.  
Elements in \text{vec} must be ordered from smaller to greater w.r.t. \text{cmp}. Using that fact, this procedure performs binary search instead of linear search.  
The optional arguments \text{start} and \text{end} are an extension to SRFI-133, and can be used to limit the range of the search in \text{start}-th element (inclusive) to \text{end}-th element (exclusive).

vector-any \( \text{pred vec1 vec2} \ldots \) 
[Function]  
[R7RS vector] \{scheme.vector\} Applies \text{pred} on each corresponding elements of \text{vec1 vec2} \ldots left to right, and as soon as \text{pred} returns non-\#f value, the procedure stops iteration and returns the value.  
If no elements that satisfy \text{pred} are found, it returns \#f.  
Vectors can have different lengths. Iteration stops at the end of the shortest.
**vector-every pred vec1 vec2 ...**

*Function*

[R7RS vector] {scheme.vector} Applies `pred` on each corresponding elements of `vec1 vec2 ...` left to right. If all the elements (when the lengths of vectors differ, the first N elements where N is the length of the shortest) satisfy `pred`, returns the last result of `pred`. If any of the elements don’t satisfy `pred`, it returns `#f` immediately without looking further.

```
(vector-every < ' #(1 2 3 4 5) ' #(2 3 4 4 5))
⇒ #f
```

```
(vector-every (^[x y] (and (real? x) (real? y) (- x y)))
 ' #(1 2 3)
 ' #(2 4 6))
⇒ -3
```

**vector-partition pred vec**

*Function*

[R7RS vector] {scheme.vector} Allocates a fresh vector of the same size as `vec`, then fill it with elements in `vec` that satisfy `pred`, followed by elements that don’t satisfy `pred`. Returns two values, the newly created vector and an exact integer of the index of the first element that doesn’t satisfy `pred` in the returned vector.

```
(vector-partition odd? ' #(1 2 3 4 5 6 7 8))
⇒ #(1 3 5 7 2 4 6 8) and 4
```

### Vector mutators

**vector-swap! vec i j**

*Function*

[R7RS vector] {scheme.vector} Swaps vector `vec`’s `i`-th and `j`-th elements. Returns unspecified value.

```
(rlet1 v (vector ' a ' b ' c ' d ' e)
 (vector-swap! v 0 2))
⇒ #(c b a d e)
```

**vector-reverse! vec :optional start end**

*Function*

[R7RS vector] {scheme.vector} Reverse the elements of `vec`. Returns an undefined value. Optional `start` and `end` arguments can limit the range of operation.

```
(rlet1 v (vector ' a ' b ' c ' d ' e)
 (vector-reverse! v 0 4))
⇒ #(d c b a e)
```

**vector-reverse-copy! target tstart source :optional sstart send**

*Function*

[R7RS vector] {scheme.vector} Like `vector-copy!`, but reverses the order of elements from `start`.

```
(rlet1 v (vector ' a ' b ' c ' d ' e)
 (vector-reverse-copy! v 2 '#(1 2)))
⇒ #(a b 2 1 e)
```

It is ok to pass the same vector to `target` and `source`; it always works even if the regions of source and destination are overlapping.

```
(rlet1 v (vector ' a ' b ' c ' d ' e)
 (vector-reverse-copy! v 1 v 1))
⇒ #(a e d c b)
```

**vector-unfold! f rvec start end seeds ...**

*Function*

[R7RS vector] {scheme.vector} Fill `rvec` starting from index `start` (inclusive) and ending at index `end` (exclusive), with the elements calculated by `f`.

**vector-unfold-right! f rvec start end seeds ...**

*Function*

[R7RS vector] {scheme.vector} Fill `rvec` starting from index `start` (inclusive) and ending at index `end` (exclusive), with the elements calculated by `f`.
The procedure \( f \) takes the number of seed values \( \text{seeds} \ldots \) plus one arguments. The first argument is the current index, followed by seed values. The same number of values as the arguments must be returned from \( f \); the first return value is used to fill the current element of \( rvec \), and the rest of the values are used as the next seed values.

The result vector is filled from left to right by \texttt{vector-unfold!}, and right to left by \texttt{vector-unfold-right!}. The return value is unspecified.

\begin{verbatim}
(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
           (vector-unfold! (^[i] (+ i 1)) rvec 1 4)
rvec)
⇒ #(a 2 3 4 e f)

(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
           (vector-unfold-right! (^[i] (+ i 1)) rvec 1 4)
rvec)
⇒ #(a 2 3 4 e f)

(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
           (vector-unfold! (^[i x] (values x (* x 2))) rvec 1 5 10)
rvec)
⇒ #(a 10 20 40 80 f)

(let1 rvec (vector 'a 'b 'c 'd 'e 'f)
           (vector-unfold! (^[i x] (values x (* x 2))) rvec 1 5 10)
rvec)
⇒ #(a 80 40 20 10 f)
\end{verbatim}

**Vector conversion**

\texttt{reverse-vector->list vec :optional start end} \hspace{1cm} \textbf{[Function]}

[R7RS vector] {scheme.vector} Same as \texttt{(reverse (vector->list vec start end))}, but more efficient.

### 10.3.3 scheme.vector.@ - R7RS uniform vectors

**scheme.vector.@** \hspace{1cm} \textbf{[Module]}

@ is actually one of u8, s8, u16, s16, u32, s32, u64, s64, f32, f64, c64 or c128. (Gauche’s \texttt{gauche.uvector} module also provides f16 and c32 vectors.)

These modules provide vectors that can hold specific range of numeric values. In Gauche we use \textit{packed} representation, meaning numbers are tightly stored in consecutive memory region.

Additionally, \texttt{scheme.vector.base} module exports basic procedures, \texttt{make-vector}, \texttt{@vector}, \texttt{@vector?}, \texttt{@vector-length}, \texttt{@vector-ref}, \texttt{@vector-set!}, \texttt{@vector->list}, \texttt{list->@vector}, etc., for @ being over all element types.

The \texttt{gauche.uvector} module is a superset of these modules, and all procedures are described there. See Section 6.13.2 [Uniform vectors], page 198, for the details.

### 10.3.4 scheme.sort - R7RS sort

**scheme.sort** \hspace{1cm} \textbf{[Module]}

Provides utilities to sort, and to work on sorted lists/vectors. This module is the same as \texttt{srfi.132}.
Gauche has built-in sort and merge procedures (see Section 6.23 [Sorting and merging], page 291). This module has a bit different API. Notably, the ordering predicate comes first than the sequence to be sorted, and the procedures dealing with vectors uniformly support start/end arguments.

This module also provide useful procedures working on sorted or partially sorted sequences.

**list-sort** \( \text{elt} < \text{lis} \)

**list-sort!** \( \text{elt} < \text{lis} \)

**list-stable-sort** \( \text{elt} < \text{lis} \)

**list-stable-sort!** \( \text{elt} < \text{lis} \)

[R7RS sort] \{scheme.sort\} Sort elements in a list \text{lis} according to the ordering predicate \( \text{elt} < \), which takes two elements from \text{lis} and returns true iff the first argument is strictly less than the second argument.

Returns a sorted list. The procedures with bang are linear update version. They are allowed, but not required, to reuse \text{lis}. The “stable” variation guarantees stable sort.

These are basically the same as Gauche’s built-in sort, sort!, stable-sort and stable-sort!, except the Gauche’s version works on any sequences and takes arguments differently. (See Section 6.23 [Sorting and merging], page 291.)

**list-sorted?** \( \text{elt} < \text{lis} \)

[R7RS sort] \{scheme.sort\} Returns true iff the list \text{list} is sorted according to the ordering predicate \( \text{elt} < \).

See also sorted? in Section 6.23 [Sorting and merging], page 291.

**list-merge** \( \text{elt} < \text{lis1 lis2} \)

**list-merge!** \( \text{elt} < \text{lis1 lis2} \)

[R7RS sort] \{scheme.sort\} Given two sorted lists \text{lis1} and \text{lis2}, returns a new sorted list according to the ordering predicate \( \text{elt} < \).

Note that **list-merge!** works in-place, that is, all the pairs in \text{lis1} and \text{lis2} are reused.

See also merge and merge! in Section 6.23 [Sorting and merging], page 291.

**vector-sort** \( \text{elt} < \text{vec :optional start end} \)

**vector-stable-sort** \( \text{elt} < \text{vec :optional start end} \)

[R7RS sort] \{scheme.sort\} Sort elements in a vector \text{vec} according to the ordering predicate \( \text{elt} < \), which takes two elements from \text{vec} and returns true iff the first argument is strictly less than the second argument. Returns a fresh sorted vector. The “stable” variation guarantees stable sort.

When the optional start and/or end arguments are given, only the portion from start (inclusive) and end (exclusive) of \text{vec} are looked at. The result vector’s length is \( \text{end} - \text{start} \). When \text{end} is omitted, the length of \text{vec} is assumed.

See also sort and stable-sort in Section 6.23 [Sorting and merging], page 291.

**vector-sort!** \( \text{elt} < \text{vec :optional start end} \)

**vector-stable-sort!** \( \text{elt} < \text{vec :optional start end} \)

[R7RS sort] \{scheme.sort\} Sort elements “in-place” in a vector \text{vec} according to the ordering predicate \( \text{elt} < \), which takes two elements from \text{vec} and returns true iff the first argument is strictly less than the second argument. Upon successful return, \text{vec}’s elements are sorted. Returns unspecified value; the caller must rely on the side effect.

Note that most of recent APIs with \( ! \) is linear updating, meaning it may or may not change the argument, and the caller must use the return value. On the other hand, vector-sort! and vector-stable-sort! are purely for side effects, and use minimal space (vector-sort!
doesn’t need extra space, while vector-stable-sort! may allocate a temporary storage up to the size of the input).

When the optional start and/or end arguments are given, only the portion from start (inclusive) and end (exclusive) of vec are sorted; other elements will remain intact. When end is omitted, the length of vec is assumed.

See also sort! and stable-sort! in Section 6.23 [Sorting and merging], page 291.

vector-sorted? elt< vec :optional start end
[Function]
[R7RS sort] {scheme.sort} Returns true iff vec between start (inclusive) and end (exclusive) is sorted according to the ordering predicate elt<. If start and/or end is/are omitted, 0 and the length of vec are assumed, respectively.

See also sorted? in Section 6.23 [Sorting and merging], page 291.

vector-merge elt< vec1 vec2 :optional start1 end1 start2 end2
vector-merge! elt< rvec vec1 vec2 :optional rstart start1 end1 start2 end2
[Function]
[R7RS sort] {scheme.sort} Merge two sorted vectors vec1 and vec2 into one vector, according to the ordering predicate elt<.

The optional argument start1 and end1 restricts vec1’s portion to be looked at, and start2 and end2 restricts vec2’s portion to be looked at. Each is integer index to the corresponding vector, and the start index is inclusive, while the end index is exclusive.

The functional version vector-merge allocates a fresh vector to hold the result, and returns it.

The side-effecting version vector-merge! uses rvec to hold the result. The procedure doesn’t return a meaningful value. The optional rstart argument specifies the index of rvec from which the result is filled; the default is 0. The length of rvec must be greater than rstart + (end1-start1) + (end2-start2).

list-delete-neighbor-dups elt= lis
list-delete-neighbor-dups! elt= lis
vector-delete-neighbor-dups elt= vec :optional start end
vector-delete-neighbor-dups! elt= vec :optional start end
[Function]
[R7RS sort] {scheme.sort} From the given list lis or vector vec, these procedures delete adjacent duplicate elements. Equivalence is checked by elt= procedure.

(list-delete-neighbor-dups eq? '(m i s s i s s i p p i))
⇒ (m i s i s i p i)

The non-destructive versions list-delete-neighbor-dups and vector-delete-neighbor-dups returns a freshly allocated list and vector, respectively.

The destructive list-delete-neighbor-dups! works in-place, reusing pairs of lis. No allocation will be done. It is a single-pass iterative algorithm, and the first cell of input is the first cell of output (except when the input is an empty list).

The destructive vector-delete-neighbor-dups! has a bit different interface. It updates vec in-place, but since we can’t change the length of the vector, it gathers the result from the beginning of the vec, then returns the next index newend of vec—that is, after calling this procedure, [start, newend) holds the result. The elements between [newend, end) will remain intact.

The optional start and end arguments limits the region of vec to be looked at.

(vector-delete-neighbor-dups eq? '='#(a a a b b c c d d e e f f) 3 10)
⇒ #(b c d e)
(let1 v '#(a a a b b c c d d e e f f)
  (cons (vector-delete-neighbor-dups! eq? v 3 10) v))
⇒ (7 . #(a a a b c d e d d e e f f))

Note: The gauche.sequence module provides neighbor duplicate deletion on generic sequences. Those procedures are implemented by the generic versions as shown below. See Section 9.31.4 [Other operations over sequences], page 512, for the details.

list-delete-neighbor-dups
delete-neighbor-dups
list-delete-neighbor-dups!
delete-neighbor-dups-squeeze!
vector-delete-neighbor-dups
delete-neighbor-dups
vector-delete-neighbor-dups!
delete-neighbor-dups!

vector-select! elt< vec k :optional start end
[R7RS sort] {scheme.sort} Select k-th smallest element in vec according to the ordering predicate elt<. K is zero based, i.e. 0 means the smallest. The optional start and end arguments limits the range of vec to be looked at, and defaulted to 0 and the length of vec, respectively. K must satisfy start <= k < end.

This procedure runs in O(n) time in average, and requires no extra storaqe. This procedure may partially modify vec.

vector-separate! elt< vec k :optional start end
[R7RS sort] {scheme.sort} Find k-th smallerst element in vec (pivot) between between start and end, according to the ordering predicate elt<, then rearrange elements between start and end so that elements smaller than the pivot comes between start and start + k, and the rest of the elements come afterwards. When omitted, start is 0 and end is the length of the vec.

This can be used as a building block for in-place divide-and-conquer algorithms. Runs in O(n) time.

vector-find-median elt< vec knil :optional mean
vector-find-median! elt< vec knil :optional mean
[R7RS sort] {scheme.sort} Find median value of elements in vec, when ordered by the ordering predicate elt<. Non-destructive version vector-find-median runs in O(n) time. The destructive version vector-find-median! is specified to leave vec sorted, so it runs in O(n log n).

1. If vec is empty, knil is returned. This is the only case knil is used.
2. If vec has odd number of elements, the element falls in the exactly the midpoint when ordered, is returned.
3. If vec has even number of elements, the two elements closest to the midpoint is chosen and passed to the procedure mean, and its result is returned. The default of mean is an arithmetic mean of numbers.

(vector-find-median < #() 0)
⇒ 0

(vector-find-median < #(78 61 19 38 51) 0)
⇒ 51
(vector-find-median < #(78 61 19 38 51 52) 0)  
⇒ 103/2

10.3.5 scheme.set - R7RS sets

scheme.set  [Module]

Sets and bags are unordered collection of Scheme values. A set doesn’t count duplicates; if you add an item which is already in a set, you still have one item of the kind. A bag counts duplicates; if you add an item which is already in a bad, you have two items of the kind.

To check whether the items are “the same”, sets and bags takes a comparator at construction time. The comparator doesn’t need to have an ordering predicate (we don’t need to order the elements) but has to have a hash function. See Section 6.2.4 [Basic comparators], page 116, for the details of comparators.

This module is originally specified as srfi.113, and then incorporated to R7RS large.

As a Gauche’s extension, sets and bags implement collection protocol (see Section 9.6 [Collection framework], page 404, for the details), and generic collection operations can be applied.

(coerce-to <list> (set eq-comparator 'a 'b 'a 'b))  
⇒ (a b) ; order may differ

(coerce-to <list> (bag eq-comparator 'a 'b 'a 'b))  
⇒ (a a b b) ; order may differ

Constructors

set comparator elt ...  [Function]

bag comparator elt ...  [Function]

[R7RS set] {scheme.set} Creates a new set and bag from given elements elt .... Given comparator will be used to compare equality of elements.

(set->list (set eq-comparator 'a 'b 'a 'b))  
⇒ (a b)

(bag->list (bag eq-comparator 'a 'b 'a 'b))  
⇒ (a a b b)

set-unfold stop? mapper successor seed comparator  [Function]

bag-unfold stop? mapper successor seed comparator  [Function]

[R7RS set] {scheme.set} Procedurally creates a set or a bag. The first three arguments, stop?, mapper and successor, are all procedures that takes one argument, the current seed value. It may be easier to know their types:

seed :: Seed
stop? :: Seed -> Boolean
mapper :: Seed -> ElementType
successor :: Seed -> Seed

The stop? procedure takes the current seed value and returns a boolean value - if it is true, iteration stops.

The mapper procedure takes the current seed value and returns an item, which is to be included in the resulting set or bag.

The successor procedure takes the current seed value and returns the next seed value.

And the seed argument gives the initial seed value.

(set->list (set-unfold (\s (= s 75)))
integer->char
(\s (+ s 1))
65
eqv-comparator))
⇒ (#\D #\H #\A #\E #\I #\J #\B #\F #\C #\G)

Predicates

set-contains? set obj [Function]
bag-contains? bag obj [Function]
   [R7RS set] {scheme.set} Check if obj is in the set or the bag.

set-empty? set [Function]
bag-empty? bag [Function]
   [R7RS set] {scheme.set} Returns #t iff the given set or bag is empty.

set-disjoint? set1 set2 [Function]
bag-disjoint? bag1 bag2 [Function]
   [R7RS set] {scheme.set} Returns #t iff the given arguments (sets or bags) don’t have common items. Both arguments must have the same comparator—otherwise an error is signaled.

Accessors

set-member set obj default [Function]
bag-member bag obj default [Function]
   [R7RS set] {scheme.set} Returns an element in the given set or bag which is equal to obj in terms of the set’s or the bag’s comparator. If no such element is found, default will be returned.

Note that the returned object doesn’t need to be “the same” as obj in a usual sense. See the following example:

(let1 s (set string-ci-comparator "abc" "def")
 (set-member s "ABC" #f))
⇒ "abc"

set-element-comparator set [Function]
bag-element-comparator bag [Function]
   [R7RS set] {scheme.set} Returns the comparator used to compare the elements for the set or the bag.

Updaters

set-adjoin set elt . . . [Function]
bag-adjoin bag elt . . . [Function]
   [R7RS set] {scheme.set} Returns a newly created set or bag that contains all the elements in the original set/bag, plus given elements. The new set/bag’s comparator is the same as the original set/bag’s one.

set-replace set elt [Function]
bag-replace bag elt [Function]
   [R7RS set] {scheme.set} Returns a newly created set/bag with the same comparator with the original set/bag, and the same elements, except that the elements equal to elt (in terms of set/bag’s comparator) is replaced by elt. If the original set/bag doesn’t contain an element equal to elt, the original one is returned.

(let ((s (set string-ci-comparator "ABC" "def")))
(set->list (set-replace s "abc")))
⇒ ("abc" "def")

(set-delete set elt ...)

(bag-delete bag elt ...)

[R7RS set] {scheme.set} Returns a newly created set or bag that has the same comparator
and the same elements in the original set/bag, except that the item which is equal to elt.

(set-delete-all set elt-list)

(bag-delete-all bag elt-list)

[R7RS set] {scheme.set} Returns a newly created set or bag with the same comparator of
the original set/bag, with the elements of the original set/bag except the ones listed in elt-list.

(set-adjoin! set elt ...)

(bag-adjoin! bag elt ...)

(set-replace! set elt)

(bag-replace! bag elt)

(set-delete! set elt ...)

(bag-delete! bag elt ...)

(set-delete-all! set elt-list)

(bag-delete-all! bag elt-list)

[R7RS set] {scheme.set} These are the linear update versions of their counterparts. It works
just like the ones without !, except that the original set/bag may be reused to produce the
result, instead of new one being allocated.

Note that it's not guaranteed that the original set/bag is modified, so you should use the
return value of them, instead of relying on the side effects.

(set-search! set elt failure success)

(bag-search! bag elt failure success)

[R7RS set] {scheme.set} Lookup-and-modify procedures. The failure and success arguments
are procedures.

First, they search an item that matches elt in the given set/bag. If an item that matches elt
is found, the success procedure is called with three arguments, as follows:

(success item update remove)

wheras, item is the matched item in the set/bag.

The update argument is a procedure that takes two arguments, as (update new-item retval).
It returns two values: (1) A set/bag that are the same as the given one except that
item is replaced with new-item, and (2) retval.

The remove argument is a procedure that takes one argument, as (remove retval). It
returns two values: (1) A set/bag that are the same as the given one except item is removed,
and (2) retval.

If an item that matches elt is not found, the failure procedure is called with two arguments,
as follows:

(failure insert ignore)

The insert argument is a procedure that takes one argument, as (insert retval). It returns
two values: (1) a set/bag that has all the items in the given one plus elt, and (2) retval.

The ignore argument is a procedure that takes one argument, as (ignore retval). It returns
two values: (1) the original set/bag, and (2) retval.

The original set/bag may be reused to produce the result by update, remove and insert
procedures.
The return values of `set-search!` and `bag-search!` is what either `success` or `failure` procedure returns. These caller-provided procedures should invoke the passed-in procedures (`update`, `remove`, `insert` or `ignore`) and return their results—which is a potentially modified set/bag (which may or may not be `eq?` to the passed one) and the `retval`.

If there are more than one item that matches `elt` in a bag, `bag-search!` only invokes `success` for the first item it finds. You can recurse into `bag-search!` in the `success` procedure to visit all matching items. It is guaranteed that `success` and `failure` procedures are tail-called.

### The whole set

- **set-size set**
  - [Function]
  - `set-size` is a procedure that returns the number of items in the set.

- **bag-size bag**
  - [Function]
  - `bag-size` is a procedure that returns the number of items in the bag.

- **set-find pred set failure**
  - [Function]
  - `set-find` returns the first item on which `pred` returns true. Since sets and bags are unordered, if there are more than one items that satisfy `pred`, you won’t know which one will be returned.

- **bag-find pred bag failure**
  - [Function]
  - `bag-find` returns the first item on which `pred` returns true. Since sets and bags are unordered, if there are more than one items that satisfy `pred`, you won’t know which one will be returned.

If there’re no items that satisfy `pred`, a thunk `failure` is tail-called.

NB: The API differs from `find` in `scheme.list` (built-in in Gauche), that `find` does not take `failure` procedure.

- **set-count pred set**
  - [Function]
  - `set-count` is a procedure that returns the number of items that satisfy `pred` in the set.

- **bag-count pred bag**
  - [Function]
  - `bag-count` is a procedure that returns the number of items that satisfy `pred` in the bag.

- **set-any? pred set**
  - [Function]
  - `set-any?` is a procedure that returns `#f` iff any item in the set satisfy `pred`, while `set-any?` always returns a boolean value (hence the question mark).

- **bag-any? pred bag**
  - [Function]
  - `bag-any?` is a procedure that returns `#f` iff any item in the bag satisfy `pred`, while `bag-any?` always returns a boolean value (hence the question mark).

- **set-every? pred set**
  - [Function]
  - `set-every?` is a procedure that returns `#t` iff every item in the set satisfy `pred`, while `set-every?` always returns a boolean value (hence the question mark).

- **bag-every? pred bag**
  - [Function]
  - `bag-every?` is a procedure that returns `#t` iff every item in the bag satisfy `pred`, while `bag-every?` always returns a boolean value (hence the question mark).

### Mapping and folding

- **set-map comparator proc set**
  - [Function]
  - `set-map` is a procedure that creates and returns a new set with the comparator `comparator`, whose items are calculated by applying `proc` to each element in the original set.

- **bag-map comparator proc bag**
  - [Function]
  - `bag-map` is a procedure that creates and returns a new bag with the comparator `comparator`, whose items are calculated by applying `proc` to each element in the original bag.

- **set-for-each proc set**
  - [Function]
  - `set-for-each` is a procedure that applies `proc` to each element in the set. The result of `proc` is ignored. Return value is undefined.

- **bag-for-each proc bag**
  - [Function]
  - `bag-for-each` is a procedure that applies `proc` to each element in the bag. The result of `proc` is ignored. Return value is undefined.
**set-fold** proc seed set

**bag-fold** proc seed bag

[R7RS set] {scheme.set} For each item in the set/bag, call proc with two arguments, an item and a seed value. What proc returns becomes the next seed value. The seed argument gives the initial seed value, and the last return value of proc will be the result of set-fold/bag-fold.

\[
(bag-fold + 0 (bag eqv-comparator 1 1 2 2 3 3 4 4)) \Rightarrow 20
\]

**set-filter** pred set

**bag-filter** pred bag

[R7RS set] {scheme.set} Returns a newly created set/bag with the same comparator of the original set/bag, and its content consists of items from the original set/bag that satisfy pred.

\[
(set->list (set-filter odd? (set eqv-comparator 1 2 3 4 5))) \Rightarrow (1 3 5)
\]

**set-remove** pred set

**bag-remove** pred bag

[R7RS set] {scheme.set} Returns a newly created set/bag with the same comparator of the original set/bag, and its content consists of items from the original set/bag that does not satisfy pred.

\[
(set->list (set-remove odd? (set eqv-comparator 1 2 3 4 5))) \Rightarrow (2 4)
\]

**set-partition** pred set

**bag-partition** pred bag

[R7RS set] {scheme.set} Returns two sets or bags, both have the same comparator of the original set or bag. The first one consists of the items from the original set/bag that satisfy pred, and the second one consists of the items that don’t.

\[
(receive (in out) (set-remove odd? (set eqv-comparator 1 2 3 4 5)))
\]

\[
(values (set->list in)
\]

\[
(set->list out))) \Rightarrow (1 3 5) \text{ and } (2 4)
\]

**set-filter!** pred set

**bag-filter!** pred bag

**set-remove!** pred set

**bag-remove!** pred bag

**set-partition!** pred set

**bag-partition!** pred bag

[R7RS set] {scheme.set} Linear update versions of their counterparts (the procedures without !). They work like their respective counterpart, but they are allowed (but not required) to reuse the original set/bag to produce the result(s).

Note that it is not guaranteed that the original set/bag is modified, so you have to use the return value(s) instead of relying on the side effects.

**Copying and conversion**

**set-copy** set

**bag-copy** bag

[R7RS set] {scheme.set} Returns a copy of the set/bag.
Chapter 10: Library modules - R7RS standard libraries

set->list set  [Function]
Bag->list bag  [Function]
  {scheme.set} Returns a list of all items in the set/bag. Since sets and bags are unordered, there's no guarantee on the order of items.

list->set comparator elt-list  [Function]
list->bag comparator elt-list  [Function]
  Creates a set or a bag with the given comparator, and the list of element. Functionally equivalent to the followings:
  (apply set comparator elt-list)
  (apply bag comparator elt-list)

list->set! set elt-list  [Function]
list->bag! bag elt-list  [Function]
  Add items in elt-list to the existing set/bag, and returns the updated set/bag. The original set/bag may or may not be modified. Functionally equivalent to the followings:
  (apply set-adjoin! set elt-list)
  (apply bag-adjoin! bag elt-list)

bag->set bag  [Function]
sset->bag set  [Function]
  Conversions between a bag and a set. Returns a newly created bag or set, respectively.
  If bag has duplicated items, bag->set coerces them to one item.

set->bag! bag set  [Function]
  Returns a bag that contains all items in set plus all items in bag, which may be modified to create the result. Both bag and set must have the same comparator.

bag->alist bag  [Function]
  Returns a list of (item . count), where item is an item in bag, and count is the number of that item in the bag.

alist->bag comparator alist  [Function]
  Creates a new bag with comparator, and fills it according to alist, which must be a list of (item . count).
  If there's duplicate items in alist, only fist one counts.

Subsets

set=? set1 set2 . . .  [Function]
bag=? bag1 bag2 . . .  [Function]
  Returns true if all sets/bags have exactly same items.

set<=? set1 set2 . . .  [Function]
set>=? set1 set2 . . .  [Function]
set<=? set1 set2 . . .  [Function]
set>=? set1 set2 . . .  [Function]
bag>=? bag1 bag2 . . .  
[R7RS set] {scheme.set} These predicates checks inclusion relationship between sets or bags. For example, (set<? set1 set2) is #t iff set1 is a proper subset of set2, and (bag>=? bag1 bag2) is #t iff bag2 is a subset (including the 'equal to' case) of bag1, etc.

If more than two arguments are given, the predicate returns #t iff every consecutive arguments satisfies the relationship.

Set theory operations

set-union set1 set2 . . .  
[Function]

bag-union bag1 bag2 . . .  
[Function]

[R7RS set] {scheme.set} Returns a newly allocated set or bag which is a union of all the sets/bags.

set-intersection set1 set2 . . .  
[Function]

bag-intersection bag1 bag2 . . .  
[Function]

[R7RS set] {scheme.set} Returns a newly allocated set or bag which is an intersection of all the sets/bags.

set-difference set1 set2 . . .  
[Function]

bag-difference bag1 bag2 . . .  
[Function]

[R7RS set] {scheme.set} Returns a newly created set or bag that contains items in set1/bag1 except those are also in set2/bag2 . . ..

(sort (set->list (set-difference (set eqv-comparator 1 2 3 4 5 6 7) (set eqv-comparator 3 5 7 9 11 13) (set eqv-comparator 4 8 16 32))))

⇒ (1 2 6)

set-xor set1 set2  
[Function]

bag-xor bag1 bag2  
[Function]

[R7RS set] {scheme.set} Returns a newly created set or bag that consists of items that are either in set1/bag1 or set2/bag2, but not in both.

(sort (set->list (set-xor (set eqv-comparator 2 3 5 7 11 13 17) (set eqv-comparator 3 5 7 9 11 13 15))))

⇒ (2 9 15 17)

set-union! set1 set2 . . .  
[Function]

bag-union! bag1 bag2 . . .  
[Function]

set-intersection! set1 set2 . . .  
[Function]

bag-intersection! bag1 bag2 . . .  
[Function]

set-difference! set1 set2 . . .  
[Function]

bag-difference! bag1 bag2 . . .  
[Function]

set-xor! set1 set2  
[Function]

bag-xor! bag1 bag2  
[Function]

[R7RS set] {scheme.set} Linear update versions of their corresponding procedures. Those procedures works like their !-less counterparts, except that they are allowed to, but not required to, reuse set1/bag1 to produce the result.

The caller should always use the returned set/bag instead of relying on the side effects.

Bag-specific procedures

bag-sum bag1 bag2 . . .  
[Function]
Chapter 10: Library modules - R7RS standard libraries

**bag-sum!**  
*bag1 bag2 . . .*  
[Function]  
[R7RS set] {scheme.set} Returns a bag that gathers all the items in given bags, counting duplicates. The functional version **bag-sum** always creates new bag to return. The linear update version **bag-sum!** is allowed to, but not required to, modify **bag1** to produce the result.

```
(sort (bag->list (bag-sum (bag eqv-comparator 1 1 2 4 5 5 6)
                       (bag eqv-comparator 3 3 5 9))))
⇒ (1 1 2 3 4 5 5 5 6 9)
```

Note the difference from **bag-union**:

```
(sort (bag->list (bag-union (bag eqv-comparator 1 1 2 4 5 5 6)
                        (bag eqv-comparator 3 3 5 9))))
⇒ (1 1 2 3 3 4 5 5 6 9)
```

**bag-product**  
*n bag*  
[Function]  

**bag-product!**  
*n bag*  
[Function]  
[R7RS set] {scheme.set} Returns a bag that contains every item as n-times many as the original bag. A fresh bag is created and returned by **bag-product**, while **bag-product!** may reuse **bag** to produce the result.

```
(sort (bag->list (bag-product 2 (bag eq-comparator 'a 'b 'r 'a))))
⇒ (a a a a b b r r)
```

**bag-unique-size**  
*bag*  
[Function]  
[R7RS set] {scheme.set} Returns the number of unique elements in **bag**.

```
(bag-unique-size (bag eqv-comparator 1 1 2 2 3 3 4))
⇒ 4
```

**bag-element-count**  
*bag elt*  
[Function]  
[R7RS set] {scheme.set} Returns the number of specified element **elt** in **bag**.

```
(bag-element-count (bag eqv-comparator 1 1 2 2 2 3 3) 2)
⇒ 3
```

**bag-for-each-unique**  
*proc bag*  
[Function]  
[R7RS set] {scheme.set} For each unique item in **bag**, calls **proc** with two arguments: The item, and the count of the item in the bag.

**bag-fold-unique**  
*proc seed bag*  
[Function]  
[R7RS set] {scheme.set} For each unique item in **bag**, calls **proc** with three arguments: The item, the count of the item, and the previous seed value. The **seed** argument provides the initial seed value; the result of **proc** is used for the next seed value, and the last result of **proc** is returned from **bag-fold-unique**.

```
(sort (bag-fold-unique acons '() (bag equal-comparator "a" "a" "b" "b" "b" "c" "d"))
   string<? car))
⇒ (("a" . 2) ("b" . 3) ("c" . 1) ("d" . 1))
```

**bag-increment!**  
*bag elt count*  
[Function]  
**bag-decrement!**  
*bag elt count*  
[Function]  
[R7RS set] {scheme.set} Linearly update **bag** to increase or decrease the count of **elt** in it by **count**, which must be an exact integer. Note that the element count won’t get below zero; if a bag has two a’s, and you call **(bag-decrement! bag 'a 100)**, you get a bag with zero a’s.
Comparators

set-comparator [Comparator]
bag-comparator [Comparator]
[R7RS comparator] {scheme.set} Comparators to be used to compare sets or bags. They
don’t provide comparison procedure, for you cannot define a total order among sets or bags.
They do provide hash functions.

10.3.6 scheme.charset - R7RS character sets

scheme.charset [Module]
Implements character set library, originally defined as SRFI-14. Note that the following
character-set procedures and pre-defined charsets are Gauche’s build-in. See Section 6.10
[Character sets], page 165.

char-set char-set? char-set-contains?
char-set-copy char-set-complement char-set-complement!
char-set:graphic char-set:printing char-set:whitespace
char-set:iso-control char-set:punctuation char-set:symbol
char-set:hex-digit char-set:blank char-set:ascii
char-set:empty char-set:full

In Gauche, the <char-set> class inherits <collection> and implements the collection pro-
tocol, so that the generic operations defined in gauche.collection can also be used (see
Section 9.6 [Collection framework], page 404).

10.3.6.1 Character-set constructors

list->char-set char-list :optional base-cs
list->char-set! char-list base-cs
[R7RS comparator] {scheme.charset} Constructs a character set from a list of characters
char-list. If base-cs is given, it must be a character set, and the characters in it are added to
the result character set. List->char-set! is allowed, but not required, to reuse base-cs to
store the result.

string->char-set s :optional base-cs
string->char-set! s base-cs
[R7RS charset] {scheme.charset} Like list->char-set and list->char-set!, but take a
list of characters from a string s.

char-set-filter pred char-set :optional base-cs
char-set-filter! pred char-set base-cs
[R7RS charset] {scheme.charset} Returns a character set containing every character c in
char-set such that (pred c) returns true. If a character set base-cs is given, its content
is added to the result. The linear update version char-set-filter! is allowed, but not
required, to modify base-cs to store the result.

ucs-range->char-set lower upper :optional error? base-cs
ucs-range->char-set! lower upper error? base-cs
[R7RS charset] {scheme.charset} Creates a character set containing every character whose
ISO/IEC 10646 UCS-4 code lies in the half-open range [lower,upper).
If the range contains unassigned codepoint, they are silently ignored.
If the range contains a valid codepoint which isn’t supported in Gauche, it is ignored when `error?` is false (default), or an error is raised when `error?` has a true value. If you compile Gauche with utf-8 native encoding (default), all valid Unicode codepoints are supported. If Gauche is compiled with other native encoding, some codepoints are not supported.

If a character set `base-cs` is given, its content is added to the result. The linear update version `ucs-range->char-set!` is allowed, but not required, to modify `base-cs` to store the result.

```scheme
[Function]
integer-range->char-set lower upper :optional error? base-cs
{scheme.charset} These are Gauche-specific procedures and not in scheme.charset. When Gauche is compiled with utf-8 native encoding (default), they are the same as ucs-range->char-set and ucs-range->char-set!, respectively. If Gauche is compiled with other native encoding, these interprets the given range in its native encoding.

Meaning of `error?` and `base-cs` are the same as ucs-char->char-set.
```

10.3.6.2 Character-set comparison

```scheme
[Function]
char-set= char-set1 ... [R7RS charset] {scheme.charset} Returns #t iff all the character sets have exactly the same members.
        (char-set=)       ⇒ #t
        (char-set= (char-set)) ⇒ #t
        (char-set= (string->char-set "cba")
          (list->char-set #\a #\b #\c))
    ⇒ #t

char-set= char-set1 ... [R7RS charset] {scheme.charset} Returns #t iff every char-set argument is a subset of the following char-sets. If no arguments are given, #t is returned.
```

```scheme
[Function]
char-set-hash char-set :optional bound [R7RS charset] {scheme.charset} Returns a non-negative exact integer as a hash value of char-set. If optional `bound` argument is given, it must be a positive integer that limits the range of the hash value, which will fall between 0 to (− bound 1), inclusive.
```

10.3.6.3 Character-set iteration

```scheme
[Function]
char-set-cursor char-set [R7RS charset] {scheme.charset} Returns an object that can point to a first character within char-set (here, ‘first’ merely means the beginning of iteration; the order of iteration is implementation-dependent and you can’t assume a specific order). The caller must treat the return value as an opaque object; it can only be used to pass as the cursor argument of char-set-ref, char-set-cursor-next and end-of-char-set?.
```
Chapter 10: Library modules - R7RS standard libraries

char-set-ref char-set cursor
[Function]
[R7RS charset] {scheme.charset} Returns a character in char-set pointed by cursor.

The cursor argument must be an object returned from char-set-cursor or char-set-cursor-next with char-set. The behavior is undefined if cursor is not a cursor created from char-set.

char-set-cursor-next char-set cursor
[Function]
[R7RS charset] {scheme.charset} Returns a new cursor for the next character pointed by cursor within char-set.

The cursor argument must be an object returned from char-set-cursor or char-set-cursor-next with char-set. The behavior is undefined if cursor is not a cursor created from char-set.

end-of-char-set? cursor
[Function]
[R7RS charset] {scheme.charset} Returns #t iff cursor points to the end of the charset.

char-set-fold kons knil char-set
[Function]
[R7RS charset] {scheme.charset} Iterate over all characters in char-set, calling the procedure kons with a character and the seed value. The return value of kons becomes the next seed value, while knil gives the first seed value. Returns the last seed value. The order of traversal isn’t specified.

char-set-unfold pred fun gen seed :optional base-char-set
[Function]
char-set-unfold! pred fun gen seed base-char-set
[Function]
[R7RS charset] {scheme.charset} Build a character set by calling fun on the seed value repeatedly. For each iteration, first pred is applied to the current seed value. If it returns true, a character set that gathers characters generated so far is returned. Otherwise, fun is called on the seed value, which must return a character. Then gen is called on the seed value to obtain the next seed value.

If base-char-set is given, a union of it and the generated characters is returned. The linear-update version char-set-unfold! may modify base-char-set to create the result.

char-set-for-each proc char-set
[Function]
[R7RS charset] {scheme.charset} Applies proc on each character in char-set. The return value of proc is discarded. Returns undefined value.

char-set-map proc char-set
[Function]
[R7RS charset] {scheme.charset} Applies proc on each character in char-set, which must return a character. Returns a new charset consists of the characters returned from proc.

10.3.6.4 Character-set query

char-set-every pred char-set
[Function]
char-set-any pred char-set
[Function]
char-set-count pred char-set
[Function]
[R7RS charset] {scheme.charset} These procedures apply pred to each character in char-set. char-set-every returns #f as soon as pred returns #f. Otherwise, it returns the result of the last application of pred.

char-set-any returns #f as soon as pred returns a true value, and the return value is the one pred returns. If pred returns #f for all characters, #f is returned.

char-set-count returns the number of times pred returns a true value.

Note that char-set can be huge (e.g. a complement of small char-set), which can make these procedures take very long.
Chapter 10: Library modules - R7RS standard libraries

[Function] char-set->list char-set
    [R7RS charset] {scheme.charset} Returns a list of each character, or a string consisting of each character, in char-set, respectively. Be careful to apply this on a large char set.

10.3.6.5 Character-set algebra

[Function] char-set-adjoin char-set char1 ...
    [R7RS charset] {scheme.charset} Returns a character set that adds char1 ... to char-set.

The linear update version char-set-adjoin! may modify char-set.

[Function] char-set-delete char-set char1 ...
    [R7RS charset] {scheme.charset} Returns a character set that removes char1 ... from char-set. It is noop if char-set doesn’t have char1 ...

The linear update version char-set-delete! may modify char-set.

[Function] char-set-union char-set ...
    [R7RS charset] {scheme.charset} Returns a character set of all characters in any one of char-set .... Without arguments, returns an empty charset.

The linear update version char-set-union! may modify char-set1.

[Function] char-set-intersection char-set ...
    [R7RS charset] {scheme.charset} Returns a character set of every character that is in all of char-set .... Without arguments, returns char-set:full.

The linear update version char-set-intersection! may modify char-set1.

[Function] char-set-difference char-set1 char-set2 ...
    [R7RS charset] {scheme.charset} Returns a character set of every character that is in char-set1 but not in any of char-set2 ....

The linear update version char-set-difference! may modify char-set1.

[Function] char-set-xor char-set ...
    [R7RS charset] {scheme.charset} With zero arguments, returns an empty charset. With one argument, it is returned as is. With two arguments, returns a character set of every character that is in either one of two sets, but not in both. With more than two arguments, it returns (char-set-xor (char-set-xor set1 set2 set3 ...)).

The linear update version char-set-xor! may modify char-set1.

[Function] char-set-diff+intersection char-set1 char-set2 ...
    [R7RS charset] {scheme.charset} Returns two values, the result of (char-set-difference char-set1 char-set2 ...) and the result of (char-set-intersection char-set1 char-set2 ...).
10.3.7 scheme.hash-table - R7RS hash tables

This module provides hash table library, originally defined as srfi.125.

Hash table provided with this module is the same as Gauche’s built-in hash table. However, SRFI-125 introduces procedures that conflict with Gauche’s original procedures, so Gauche provides those procedures built-in but under aliases. See Section 6.14.1 [Hashtables], page 206, for the built-in hash table procedures.

With this module, procedures are provided as defined in R7RS. Use this module when you’re writing portable code.

SRFI-125 also defines compatibility procedures with SRFI-69, although saying they’re deprecated. Those deprecated procedures are supported in this module, too.

The following procedures are the same as Gauche’s built-in.

- hash-table?
- hash-table-contains?
- hash-table-exists?
- hash-table-empty?
- hash-table=?
- hash-table-mutable?
- hash-table-ref
- hash-table-ref/default
- hash-table-set!
- hash-table-update!/default
- hash-table-clear!
- hash-table-size
- hash-table-keys
- hash-table-values
- hash-table-copy
- hash-table-empty-copy
- hash-table->alist
- hash-table-union!
- hash-table-intersection!
- hash-table-difference!
- hash-table-xor!

See Section 6.14.1 [Hashtables], page 206, for the description of those procedures.

The following procedures are also provided as Gauche’s built-in, but with -r7 suffix.

- hash-table
- hash-table-delete!
- hash-table-intern!
- hash-table-update!
- hash-table-pop!
- hash-table-find
- hash-table-count
- hash-table-map
- hash-table-for-each
- hash-table-map!
- hash-table-map->list
- hash-table-prune!

make-hash-table comparator arg . . .

This enhances built-in make-hash-table with the second form, that takes two procedures instead of one comparator, as SRFI-69.

In the SRFI-69 form, equal-proc is a procedure taking two keys to see if they are the same, and hash-proc is a procedure taking a key to calculate its hash value (nonnegative fixnum). The compatibility form is deprecated and should be avoided in the new code.

The optional arg . . . are implementation-dependent, and specify various properties of the created hash table. They are ignored in Gauche.

hash-table cmpr key value . . .

This is the same as built-in hash-table-r7 (see Section 6.14.1 [Hashtables], page 206).

Construct a new hash table with key-comparator cmpr. It is populated by key value . . . , which is a list with keys and values appear alternatively. It is an error if the length of key-value list is not even.

Note that SRFI-125 defines this procedure to return an immutable hash table if the implementation supports one. Gauche doesn’t provide immutable hash tables (we do have immutable map instead, see Section 12.17 [Immutable map], page 834), but when you’re writing a portable program, be careful not to modify the table returned by this procedure.
alist->hash-table  alist cmpr arg . . .  [Function]
alist->hash-table  alist equal-proc hash-proc cmpr arg . . .  [Function]

[R7RS hash-table] {scheme.hash-table} This enhances built-in alist->hash-table with the second form, that takes two procedures instead of one comparator, as SRFI-69.

In the SRFI-69 form, equal-proc is a procedure taking two keys to see if they are the same, and hash-proc is a procedure taking a key to calculate its hash value (nonnegative fixnum).

The compatibility form is deprecated and should be avoided in the new code.

The optional arg . . . are implementation-dependent, and specify various properties of the created hash table. They are ignored in Gauche.

hash-table-unfold  p f g seed comparator :optional arg . . .  [Function]

[R7RS hash-table] {scheme.hash-table} Same as Gauche’s built-in hash-table-unfold, except that this allows optional arguments, which are ignored. They are implementation-specific parameters to tune the created hash tables. See Section 6.14.1 [Hashtables], page 206, for the detailed description.

hash-table-delete!  ht key . . .  [Function]

[R7RS hash-table] {scheme.hash-table} This is the same as built-in hash-table-delete!-r7.

Deletes entries associated with the given keys from the table ht. It is ok if ht doesn’t have key. Returns the number of entries that are actually deleted.

It differs from built-in hash-table-delete! in two points: The built-in one can take exactly one key, and returns a boolean indicating if the entry is actually deleted.

hash-table-intern!  ht key failure  [Function]

[R7RS hash-table] {scheme.hash-table} This is the same as built-in hash-table-intern!-r7.

Search key in ht. If it is found, returns the associated value. If it is not found, call failure without arguments, and insert a new entry associating key and the value failure returns, and returns that new value.

hash-table-update!  ht key updater :optional failure success  [Function]

[R7RS hash-table] {scheme.hash-table} This is the same as built-in hash-table-update!-r7. It takes different optional arguments from built-in hash-table-update!.

Updater is a procedure that takes one argument, failure is a thunk, and success is a procedure that takes one argument.

Works the same as follows, except maybe more efficiently.

(hash-table-set! ht key (updater (hash-table-ref ht key failure success)))

(hash-table-pop!  ht  [Function]

[R7RS hash-table] {scheme.hash-table} This is the same as built-in hash-table-pop!-r7.

It is a completely different procedure as built-in hash-table-pop!.

Removes an arbitrary entry in the hash table ht, and returns the removed entry's key and value as two values.

If ht is empty, an error is signalled.

(hash-table-find  proc ht failure  [Function]

[R7RS hash-table] {scheme.hash-table} This is the same as built-in hash-table-find-r7.

It takes different arguments from built-in hash-table-find.

Calls proc with a key and a value of each entry in ht, until proc returns non-false value. If proc returns non-false value, hash-table-find immediately returns it. If proc returns #f for all entries, calls a thunk failure and returns its result.
hash-table-count\[\text{proc}\]\[\text{ht}\]

[R7RS hash-table] \{\text{scheme.hash-table}\} This is the same as built-in \text{hash-table-count-r7}.

Calls \text{proc} with a key and a value of each entry in \text{ht}, and returns the number of times when \text{proc} returned true.

hash-table-map\[\text{proc}\]\[\text{cmptr}\]\[\text{ht}\]

[R7RS hash-table] \{\text{scheme.hash-table}\} This is the same as built-in \text{hash-table-map-r7}.

This is different from built-in \text{hash-table-map}.

Creates a fresh hashtable with a key comparator \text{cmptr}, then populate it by inserting the key and the result of applying \text{proc} on the value of each entry in \text{ht}.

hash-table-map!\[\text{proc}\]\[\text{ht}\]

[R7RS hash-table] \{\text{scheme.hash-table}\} This is the same as built-in \text{hash-table-map!-r7}.

Calls \text{proc} on the key and value of each entry in \text{ht}, and update the entry of the key with the result of \text{proc}.

hash-table-map->list\[\text{proc}\]\[\text{ht}\]

[R7RS hash-table] \{\text{scheme.hash-table}\} This is the same as built-in \text{hash-table-map->list-r7}, and same as built-in \text{hash-table-map} (not the \text{scheme.hash-table}’s \text{hash-table-map}) except the order of the arguments.

Apply \text{proc} on a key and a value of each entry in \text{ht}, in arbitrary order, and returns a list of results.

hash-table-for-each\[\text{proc}\]\[\text{ht}\]

hash-table-for-each\[\text{ht}\]\[\text{proc}\]

[R7RS hash-table] \{\text{scheme.hash-table}\} Apply \text{proc} on a key and a value of each entry in \text{ht}. The result of \text{proc} is discarded. Returns an unspecified value.

This procedure allows arguments in both order for the compatibility— the first way is the \text{scheme.hash-table} recommended one, which is the same as built-in \text{hash-table-for-each-r7}, and the latter way is compatible with SRFI-69, which is the same as built-in \text{hash-table-for-each}.

It is unfortunate that this compatibility thing is extremely confusing; especially in Gauche, you can make anything applicable, so the distinction between procedures and other objects is blurred.

We recommend that you stick to one way or another within a module; if your module uses built-in interface, use \text{(hash-table-for-each ht proc)}. If your module imports \text{scheme.hash-table}, use \text{(hash-table-for-each proc ht)}.

hash-table-fold\[\text{proc}\]\[\text{seed}\]\[\text{ht}\]

hash-table-fold\[\text{ht}\]\[\text{proc}\]\[\text{seed}\]

[R7RS hash-table] \{\text{scheme.hash-table}\} The \text{proc} argument takes three arguments, a key, a value, and the current seed value. The procedure applies \text{proc} for each entry in \text{ht}, using \text{seed} as the first seed value, and using the previous result of \text{proc} as the subsequent seed value. Returns the result of the last call of \text{seed}.

This procedure allows arguments in both order for the compatibility— the first way is the \text{scheme.hash-table} recommended one, which is the same as built-in \text{hash-table-fold-r7}, and the latter way is compatible with SRFI-69, which is the same as built-in \text{hash-table-fold}.

It is unfortunate that this compatibility thing is extremely confusing. We recommend that you stick to one way or another within a module; if your module uses built-in interface, use the second interface. If your module is for portable R7RS code, use the first interface.
Chapter 10: Library modules - R7RS standard libraries

hash-table-prune! proc ht
[Function]  
[R7RS hash-table] {scheme.hash-table} This is the same as built-in hash-table-prune!-r7.

Apply proc on a key and a value of each entry in ht, and deletes the entry if proc returns a true value. This procedure returns an unspecified value.

hash-table-merge! ht1 ht2  
[Function]  
[R7RS hash-table] {scheme.hash-table} Deprecated. This is the same as hash-table-union!, and provided just for the compatibility with SRFI-69.

hash obj :optional ignore
[Function]
string-hash obj :optional ignore
[Function]
string-ci-hash obj :optional ignore
[Function]
hash-by-identity obj :optional ignore
[Function]  
[R7RS hash-table] {scheme.hash-table} Provided for the compatibility with SRFI-69, and are deprecated.

The first three are the same as built-in default-hash, string-hash, and string-ci-hash, except that these accept an optional second argument, which is ignored. Note that hash-by-identity is also defined as the same as default-hash except the ignored optional second argument, per SRFI-125, although the name suggests that it would work as if eq-hash.

Do not use these procedures in the new code; you can use comparators instead (default-comparator, string-comparator, string-ci-comparator and eq-comparator, see Section 6.2.4.3 [Predefined comparators], page 119). If you do need hash function, you should still avoid hash and hash-by-identity, and use default-hash and eq-hash instead.

hash-table-equivalence-function ht  
[Function]
hash-table-hash-function ht  
[Function]  
[R7RS hash-table] {scheme.hash-table} Provided for the compatibility with SRFI-69, and are deprecated.

Returns the equivalence function and hash function of a hash table ht.

Note that SRFI-69’s hash function takes an optional bound argument; so, if you build ht with a comparator or a hash function that does not take optional argument, hash-table-hash-function wraps the real hash function with a procedure that allows optional bound argument.

For the introspection, we recommend to use built-in hash-table-comparator (see Section 6.14.1 [Hashtables], page 206).

10.3.8 scheme.ilmist - R7RS immutable lists

scheme.ilmist
[Module]

This module provides a set of procedures that handles immutable pairs and lists.

The standard allows an implementation to have mutable pairs and immutable pairs separately, so it defines immutable version of most scheme.list procedures. In Gauche, mutable pairs and immutable pairs are both abstract “pairs”, and all procedures that accesses pairs (without modifying them) works seamlessly on both kind.

Consequently, the following procedures are just aliases of their non-immutable versions (just remove i prefix from them). Note that these procedures and variables in Gauche do not reject if the input is mutable pairs/lists, but such usage may not be portable (see Section 6.6.2 [Mutable and immutable pairs], page 141).

proper-ilmist? ilmist? dotted-ilmist? not-ipair?
Chapter 10: Library modules - R7RS standard libraries

This module provides an alternative datatype for pairs and lists, which are immutable and allows $O(\log n)$ random access, while maintaining $O(1)$ car and cdr operations. This is originally defined as srfi.101. We call this datatype rlist in the following explanation.

The srfi allows a Scheme implementation to adopt rlist as the built-in pairs and lists, so the procedure names are duplicated from the Scheme primitives. If you use this module, you might want to import with prefix (e.g. `(use scheme.rlist :prefix rlist:)`).

In Gauche, we implement rlist on top of skew-list (see Section 12.23 [Skew binary random-access lists], page 851). The main difference is that skew-list only allows proper lists, while rlist allows improper lists (the cdr of the last pair can be an arbitrary object).

However, having improper lists makes things a lot complicated. If you just need a list with fast random access, you want to use skew-list. Use rlist only if you need to deal with improper lists.

The following procedures behave just like the built-in counterparts, except they take/return rlists instead of ordinary pairs and lists. (NB: list-ref and list-tail here doesn’t take the optional fallback argument, for they are Gauche’s extension).

```scheme
pair? cons car cdr
caar cadr cdar cddr
caaar caadr cadar caddr
cdaar cdadr cddar cdddrr
caaaar caaadr caadrr caaddrr
cdaaar cdaadr cdadar cdaddrr
cddaar cddadr cddadar cddddrr
null? list? list make-list
length append reverse list-tail
```

10.3.9 scheme.rlist - R7RS random-access lists

This module provides an alternative datatype for pairs and lists, which are immutable and allows $O(\log n)$ random access, while maintaining $O(1)$ car and cdr operations. This is originally defined as srfi.101. We call this datatype rlist in the following explanation.

The srfi allows a Scheme implementation to adopt rlist as the built-in pairs and lists, so the procedure names are duplicated from the Scheme primitives. If you use this module, you might want to import with prefix (e.g. `(use scheme.rlist :prefix rlist:)`).

In Gauche, we implement rlist on top of skew-list (see Section 12.23 [Skew binary random-access lists], page 851). The main difference is that skew-list only allows proper lists, while rlist allows improper lists (the cdr of the last pair can be an arbitrary object).

However, having improper lists makes things a lot complicated. If you just need a list with fast random access, you want to use skew-list. Use rlist only if you need to deal with improper lists.

The following procedures behave just like the built-in counterparts, except they take/return rlists instead of ordinary pairs and lists. (NB: list-ref and list-tail here doesn’t take the optional fallback argument, for they are Gauche’s extension).

```scheme
pair? cons car cdr
caar cadr cdar cddr
caaar caadr cadar caddr
cdaar cdadr cddar cdddrr
caaaar caaadr caadrr caaddrr
cdaaar cdaadr cdadar cdaddrr
cddaar cddadr cddadar cddddrr
null? list? list make-list
length append reverse list-tail
```
list-ref map for-each

This module also exports a syntax quote, which denotes the literal rlist. (quote (a (b c))) becomes a literal rlist consists of elements a, and an rlist consists of b and c.

Note that if you import scheme.rlist without suffix, the shorthand notation ’(a (b c)) refers to scheme.rlist#quote, instead of built-in quote.

list-set rlist k obj

[Function]
[R7RS rlist] {scheme.rlist} Returns a new rlist which is the same as rlist except that its k-th element is replaced with obj. This is O(log n) operation where n is the length of rlist.

list-ref/update rlist k proc

[Function]
[R7RS rlist] {scheme.rlist} Returns two values, the result of (list-ref rlist k) and the result of (list-set rlist k (proc (list-ref rlist k))), but may be more efficient.

random-access-list->linear-access-list rlist

[Function]
[R7RS rlist] {scheme.rlist} Convert a proper rlist to an ordinary list. An error is thrown when rlist is not proper. The conversion is only done in the “spine” of rlist; that is, if rlist is nested, only the outermost rlist is converted.

linear-access-list->random-access-list list

[Function]
[R7RS rlist] {scheme.rlist} Convert a proper ordinary list to an rlist. An error is thrown when list is not proper. The conversion is only done in the “spine” of list; that is, if list is nested, only the outermost list is converted.

10.3.10 scheme.ideque - R7RS immutable deques

scheme.ideque

[Module]
This module provides a functional double-ended queue (deque, pronounced as “deck”), with amortized O(1) access of queue operations on either end.

It also serves as a convenient bidirectional list structures in a sense that operations from the end of the list is just as efficient as the ones from the front.

Note: If you don’t need immutability and wants space-efficient deque, you can also use data.ring-buffer as a deque (see Section 12.22 [Ring buffer], page 849).

This module was originally defined as srfi.134, then became a part of R7RS large.

Gauche’s data.ideque is a superset of this module. See Section 12.16 [Immutable deques], page 833.

ideque element ...  

[Function]
[R7RS ideque] {scheme.ideque} Returns an ideque with the given elements.

ideque-unfold p f g seed

[Function]
[R7RS ideque] {scheme.ideque} Construct an ideque in the same way as unfold (see Section 10.3.1 [R7RS lists], page 593), that is, a predicate p is applied to a current state value; if it returns true, the iteration stops. Otherwise, an item generator function f is applied to the current state value, and its value becomes an element of the ideque. Then a next state function g is applied to the current state value, and its result becomes the next state value, and we iterate. The result of f is gathered in order, and becomes the content of the returned ideque. The seed argument specifies the initial state value.

(idique->list
  (ideque-unfold (cut > <> 10) square (cut + <> 1) 0))
⇒ (0 1 4 9 16 25 36 49 64 81 100)
ideque-unfold-right \( p \ f \ g \ seed \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Similar to ideque-unfold, but the generated elements are stored in the reverse order in the resulting ideque.

\[
\text{ideque->list} \\
\text{(ideque-unfold-right (cut > <> 10) square (cut + <> 1) 0))} \\
\Rightarrow (100 \ 81 \ 64 \ 49 \ 36 \ 25 \ 16 \ 9 \ 4 \ 1 \ 0)
\]

ideque-tabulate size init  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Create an ideque of length size, whose \( i \)-th element is computed by \( \text{(init } i \text{)} \).

\[
\text{(ideque->list (ideque-tabulate 11 square))} \\
\Rightarrow (0 \ 1 \ 4 \ 9 \ 16 \ 25 \ 36 \ 49 \ 64 \ 81 \ 100)
\]

ideque? \( obj \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns true iff \( obj \) is an ideque.

ideque-empty? \( idq \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns true iff an ideque \( idq \) is empty.

ideque-add-front \( idq \ x \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns an ideque with an item \( x \) added to the front or back of an ideque \( idq \), respectively. These are O(1) operations.

ideque-front \( idq \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns the front or back element of an ideque \( idq \), respectively. An error is thrown if \( idq \) is empty. These are O(1) operations.

ideque-reverse \( idq \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns an ideque with the elements of \( idq \) in reverse order. This is O(1) operation.

ideque= elt = idq . . .  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns \#t iff all ideques are of the same length, and each corresponding element compares equal by \( \text{elt=} \), which is called with two arguments, one element from \( i \)-th ideque and another element from \( (i+1) \)-th ideque.

If zero or one ideque is given, \( \text{elt=} \) won’t be called at all and this procedure returns \#t.

ideque-ref \( idq \ n \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns \( n \)-th element of \( idq \). An error is signaled if \( n \) is out of range.

ideque-take \( idq \ n \)  

[Function]  
[R7RS ideque]  \{scheme.ideque\} Returns an ideque that has first or last \( n \) elements of \( idq \), respectively. An error is signaled if \( n \) is greater than the length of \( idq \).
ideque-drop idq n
[Function]
[72x726]ideque-drop-right idq n
[Function]
[R7RS ideque] {scheme.ideque} Returns an ideque that has elements of idq with first or last n elements removed, respectively. An error is signaled if n is greater than the length of idq.

ideque-split-at idq n
[Function]
[R7RS ideque] {scheme.ideque} Returns two ideques, the first one is (ideque-take idq n) and the second one is (ideque-drop idq n). An error is signaled if n is greater than the length of idq.

ideque-length idq
[Function]
[R7RS ideque] {scheme.ideque} Returns the length of idq.

ideque-append idq ...
[Function]
[R7RS ideque] {scheme.ideque} Returns an ideque that concatenates contents of given ideques.

ideque-zip idq idq2 ...
[Function]
[R7RS ideque] {scheme.ideque} Returns an ideque whose i-th element is a list of i-th elements of given ideques. If the length of given ideques differ, the result terminates at the shortest ideque.

(ideque->list
 (ideque-zip (ideque 'a 'b 'c) (ideque 1 2) (ideque 'x 'y 'z)))
⇒ ((a 1 x) (b 2 y))

ideque-map proc idq ...
[Function]
[R7RS ideque] {scheme.ideque} Applies proc to each element of idq, and returns an ideque contains the results in order. The dynamic order of calls of proc is unspecified. This is O(n) operation.

Note that R7RS ideque only supports single idq argument. It is Gauche’s extension to allow more than one idq arguments; in that case, proc gets as many arguments as the given ideques, each argument taken from each ideque. The length of the result ideque is limited to the shortest length of the input ideques.

(ideque->list (ideque-map square (ideque 1 2 3 4 5)))
⇒ (1 4 9 16 25)

(ideque->list (ideque-map + (ideque 1 2 3 4 5) (ideque 2 2 2)))
⇒ (11 12 13)

ideque-filter-map proc idq ...
[Function]
[R7RS ideque] {scheme.ideque} Applies proc to each element of idq, and returns an ideque contains the results in order, except #f. The dynamic order of calls of proc is unspecified.

Note that R7RS ideque only supports single idq argument. It is Gauche’s extension to allow more than one idq arguments; in that case, proc gets as many arguments as the given ideques, each argument taken from each ideque. The input is read until any of the idq reaches at the end.

ideque-for-each proc idq ...
[Function]
ideque-for-each-right proc idq ...
[Function]
[R7RS ideque] {scheme.ideque} Applies proc to each element of idq from left to right (ideque-for-each) or right to left (ideque-for-each-right). Results of proc are discarded.

Returns an unspecified value.

Note that R7RS ideque only supports single idq argument. If more than one ideques are given, proc gets as many arguments as the given ideques, each argument taken from each ideque.
ideque-for-each takes each ideque’s elements from leftmost ones, while ideque-for-each-right takes each ideque’s elements from rightmost ones. Both ends when the shortest ideque is exhausted.

```
(ideque-for-each ($ display $ list $)
    (ideque 'a 'b 'c 'd 'e)
    (ideque 1 2 3))
⇒ prints (a 1)(b 2)(c 3)
```

```
(ideque-for-each-right ($ display $ list $)
    (ideque 'a 'b 'c 'd 'e)
    (ideque 1 2 3))
⇒ prints (e 3)(d 2)(c 1)
```

ideque-fold proc knil idq ... [Function]
ideque-fold-right proc knil idq ... [Function]
    [R7RS ideque] {scheme.ideque}

ideque-append-map proc idq ... [Function]
    [R7RS ideque] {scheme.ideque}

ideque-filter pred idq [Function]
ideque-remove pred idq [Function]
    [R7RS ideque] {scheme.ideque}

ideque-partition pred idq [Function]
    [R7RS ideque] {scheme.ideque}

ideque-find pred idq :optional failure [Function]
ideque-find-right pred idq :optional failure [Function]
    [R7RS ideque] {scheme.ideque}

ideque-take-while pred idq [Function]
ideque-take-while-right pred idq [Function]
    [R7RS ideque] {scheme.ideque}

ideque-drop-while pred idq [Function]
ideque-drop-while-right pred idq [Function]
    [R7RS ideque] {scheme.ideque}

ideque-span pred idq [Function]
ideque-break pred idq [Function]
    [R7RS ideque] {scheme.ideque}

ideque-any pred idq ... [Function]
ideque-every pred idq ... [Function]
    [R7RS ideque] {scheme.ideque}

ideque->list idq [Function]
list->ideque list [Function]
    [R7RS ideque] {scheme.ideque}

ideque->generator idq [Function]
generator->ideque gen [Function]
    [R7RS ideque] {scheme.ideque}
10.3.11 scheme.text - R7RS immutable texts

**scheme.text** [Module]

This module provides text type, an immutable string with O(1) random access. This is originally defined in SRFI-135.

In Gauche, a string with index access generally takes O(n) because of multi-byte representation; however, if the string is ASCII-only or you precalculate string indexes, index access becomes O(1) (see Section 6.11.6 [String indexing], page 176). So, a text in Gauche is just an immutable indexed string. It satisfies string?, too.

Since text is not disjoint from string in Gauche, textual type mentioned in this srfi is equivalent to a string type, and many textual-* procedures are just aliases to the corresponding string-* procedures.

However, if you’re writing portable code, keep it in mind that some implementation may have disjoint text and string types.

**text? obj** [Function]

[R7RS text] {scheme.text} Returns #t iff obj is a text, which is an immutable indexed string in Gauche. Note that a text type is not disjoint from a string in Gauche.

**textual? obj** [Function]

[R7RS text] {scheme.text} Returns #t iff obj is either a text or a string. In Gauche, this is just an alias of string?.

**textual-null? obj** [Function]

[R7RS text] {scheme.text} Returns #t iff obj is an empty string/text. In Gauche this is just an alias of string-null?.

**textual-every** pred textual :optional start end [Function]

**textual-any** pred textual :optional start end [Function]

[R7RS text] {scheme.text} Like string-every and string-any, but can work on both texts and strings. In Gauche, these are just aliases of string-every and string-any.

**make-text** len char [Function]

[R7RS text] {scheme.text} In Gauche it is same as make-string, except that the returned string is immutable and indexed, and you can’t omit char.

**text** char ... [Function]

[R7RS text] {scheme.text} Like string, but returned string is immutable and indexed.

**text-tabulate** proc len [Function]

[R7RS text] {scheme.text} Like string-tabulate, but returned string is immutable and indexed (see Section 6.11.3 [String constructors], page 172).

**text-unfold** p f g seed :optional base make-final [Function]

**text-unfold-right** p f g seed :optional base make-final [Function]

[R7RS text] {scheme.text} Like string-unfold and string-unfold-right, but returned string is immutable and indexed (see Section 6.11.3 [String constructors], page 172). The mapper procedure f may return a string instead of a character.

**text-length** text [Function]

[R7RS text] {scheme.text} Returns the length of text. An error is signaled if text is not a text.

**text-ref** text index [Function]

[R7RS text] {scheme.text} Returns index-th character of the text. Guaranteed to be O(1). An error is signaled if text is not a text.
textual-length textual
[Function]
[R7RS text] {scheme.text} Returns the length of a text or a string. This is just an alias of string-length in Gauche.

textual-ref textual index
[Function]
[R7RS text] {scheme.text} Returns index-th character of the text. This is just an alias of string-ref in Gauche.

textual->text textual
[Function]
[R7RS text] {scheme.text} Takes a string, and returns an immutable, indexed string of the same content. If textual is already a such string, it is returned as is.

textual->string textual :optional start end
[Function]
textual->vector textual :optional start end
[Function]
textual->list textual :optional start end
[Function]
[R7RS text] {scheme.text} Converts a textual to a fresh mutable string, a vector and a list, respectively. These are aliases of string-copy, string->vector and string->list in Gauche.

string->text string :optional start end
vector->text char-vector :optional start end
list->text char-list :optional start end
[Function]
[R7RS text] {scheme.text} Convert a string, a vector of characters, and a list of characters to an immutable indexed string. If an immutable indexed string is given to string->text without start/end arguments, the input is returned as is.

reverse-list->text char-list
[Function]
[R7RS text] {scheme.text} Same as (list->text (reverse char-list)) but maybe more efficient.

textual->utf8 textual :optional start end
[Function]
[R7RS text] {scheme.text} Returns a bytevector (u8vector) that contains utf8 encoding of the input textual. In Gauche, it is the same as string->utf8 (see Section 9.37.1 [Unicode transfer encodings], page 550).

textual->utf16 textual :optional start end
[Function]
[R7RS text] {scheme.text} Returns a bytevector (u8vector) that contains utf16 encoding of the input textual, with BOM attached, and in the native byteorder. In Gauche, it is the same as (string->utf16 textual (native-endian) #t [start end]) (see Section 9.37.1 [Unicode transfer encodings], page 550).

textual->utf16be textual :optional start end
textual->utf16le textual :optional start end
[Function]
[R7RS text] {scheme.text} Returns a bytevector that contains utf16be and utf16le encoding of the input textual. No BOM is attached. In Gauche, they are the same as (string->utf16 textual ’big-endian #f [start end]) and (string->utf16 textual ’little-endian #t [start end]) (see Section 9.37.1 [Unicode transfer encodings], page 550).

utf8->text bytevector :optional start end
[Function]
[R7RS text] {scheme.text} Converts a utf8 octet sequence stored in bytevector (u8vector) to a text. If the octet sequence begins with BOM sequence, it is interpreted as a character U+FEFF. Optional start/end arguments limits the input range of bytevector.
If the input contains an invalid utf-8 sequence, and Gauche’s native encoding is utf-8, it is replaced by a unicode replacement character U+FFFD. (NB: SRFI-135 says it is an error.)
utf16->text bytevector :optional start end          [Function]
utf16be->text bytevector :optional start end       [Function]
utf16le->text bytevector :optional start end       [Function]

[R7RS text] {scheme.text} Converts a utf16 octet sequence stored in bytevector to a text. For utf16->text, the sequence may begin with BOM, in which case it determines the endianness. Otherwise, platform’s native endianness is assumed. For utf16be->text and utf16le->text, the input is assumed to be in UTF16BE/UTF16LE respectively; if it begins with BOM it is treated as a character U+FEFF.

Optional start/end arguments limits the input range of bytevector. If the length of input (end - start, if the range is limited) isn’t even, an error is thrown.

If the input contains an invalid utf-16 sequence (unpaired surrogates), it is replaced by a unicode replacement character U+FFFD. (NB: SRFI-135 says it is an error.)

subtext text start end                        [Function]
subtextual textual start end                  [Function]

[R7RS text] {scheme.text} Returns a text between start-th (inclusive) and end-th (exclusive) characters in the input.

textual-copy textual :optional start end       [Function]

[R7RS text] {scheme.text} Returns a copy of textual, optionally limited between start/end, as a text.

SRFI-135 specifies that even the input is a text, the returned one must be freshly allocated (as opposed to subtextual, which is allowed to return the result sharing the input). In Gauche, string body is immutable anyway, so it isn’t a big deal.

textual-take textual nchars                    [Function]
textual-drop textual nchars                    [Function]
textual-take-right textual nchars              [Function]
textual-drop-right textual nchars              [Function]
textual-pad textual len :optional char start end [Function]
textual-pad-right textual len :optional char start end [Function]
textual-trim textual :optional pred start end   [Function]
textual-trim-right textual :optional pred start end [Function]
textual-trim-both textual :optional pred start end [Function]

[R7RS text] {scheme.text} In Gauche, these are the same as corresponding string operations (string-take, string-drop, string-take-right, string-drop-right, string-pad, string-pad-right, string-trim, string-tirm-right, and string-trim-both), except that the returned string is always immutable and indexed. See Section 11.6.4 [SRFI-13 String selection], page 694, for these string procedures.

textual-replace textual1 textual2 start1 end1 :optional start2 end2         [Function]

[R7RS text] {scheme.text} In Gauche, this is same as string-replace, except that the returned string is always immutable and indexed. See Section 11.6.12 [SRFI-13 Other string operations], page 700, for the details.

textual=? textual1 textual2 textual3 . . .         [Function]
textual<=? textual1 textual2 textual3 . . .        [Function]
textual>=? textual1 textual2 textual3 . . .        [Function]
textual>?> textual1 textual2 textual3 . . .         [Function]

[R7RS text] {scheme.text} EN In Gauche, these are just aliases of built-in string=?, string<?, string<==?, string>?, string>?> and string>=?, respectively. See Section 6.11.8 [String comparison], page 177, for the details.
textual-ci=? textual1 textual2 textual3 ...  [Function]
textual-ci<? textual1 textual2 textual3 ...  [Function]
textual-ci>? textual1 textual2 textual3 ...  [Function]
textual-ci<=? textual1 textual2 textual3 ...  [Function]
textual-ci=>? textual1 textual2 textual3 ...  [Function]

[R7RS text] \{scheme.text\} In Gauche, these are just aliases of the \textit{unicode version} of \textit{string-ci=?}, \textit{string-ci<?}, \textit{string-ci>?}, \textit{string-ci<=?} and \textit{string-ci=>?}, respectively. See Section 9.37.3 \[Full string case conversion\], page 554, for the details.

textual-prefix-length textual1 textual2 :optional start1 end1 start2 end2  [Function]
textual-suffix-length textual1 textual2 :optional start1 end1 start2 end2  [Function]
textual-prefix? textual1 textual2 :optional start1 end1 start2 end2  [Function]
textual-suffix? textual1 textual2 :optional start1 end1 start2 end2  [Function]

[R7RS text] \{scheme.text\} In Gauche, these are just aliases of \textit{srfi.13}'s \textit{string-prefix-length}, \textit{string-suffix-length}, \textit{string-prefix?}, and \textit{string-suffix?}, respectively. See Section 11.6.6 \[SRFI-13 String prefixes & suffixes\], page 696, for the details.

textual-index textual pred :optional start end  [Function]
textual-index-right textual pred :optional start end  [Function]
textual-skip textual pred :optional start end  [Function]
textual-skip-right textual pred :optional start end  [Function]
textual-contains textual-haystack textual-needle :optional start1 end1 start2 end2  [Function]
textual-contains-right textual-haystack textual-needle :optional start1 end1 start2 end2  [Function]

[R7RS text] \{scheme.text\} In Gauche, these are just aliases of \textit{srfi.13}'s \textit{string-index}, \textit{string-index-right}, \textit{string-skip}, \textit{string-skip-right}, \textit{string-contains}, and \textit{string-contains-right}, respectively. See Section 11.6.7 \[SRFI-13 String searching\], page 697, for the details.

textual-upcase textual  [Function]
textual-downcase textual  [Function]
textual-foldcase textual  [Function]
textual-titlecase textual  [Function]

[R7RS text] \{scheme.text\} In Gauche, these are the same as \textit{string-upcase}, \textit{string-downcase}, \textit{string-foldcase} and \textit{string-titlecase} of \texttt{gauche.unicode}, respectively, except that the result is always an immutable and indexed string. See Section 9.37.3 \[Full string case conversion\], page 554, for the details.

textual-append textual ...  [Function]
textual-concatenate textual-list  [Function]

[R7RS text] \{scheme.text\} In Gauche, these are the same as \textit{string-append} and \textit{string-concatenate} respectively, except that the result is always an immutable and indexed string.

textual-concatenate-reverse args :optional final-textual end  [Function]

[R7RS text] \{scheme.text\} The \textit{args} must be a list of textuals (same as strings in Gauche). As to the optional arguments, \textit{final-textual} must be a textual, and \textit{end} must be a nonnegative exact integer, if provided.

Without optional arguments, this is the same as \textit{string-concatenate-reverse} of \textit{srfi.13} (see Section 11.6.9 \[SRFI-13 String reverse & append\], page 698, except that the result is always an immutable and indexed string.
If final-textual is provided, it is appended to the last. If end is also provided, (subtext final-textual 0 end) is appended to the last.

textual-join textual-list optional delimiter grammar

[Function]
[R7RS text] {scheme.text} This is the same as string-join except that the result is always an immutable and indexed string (see Section 6.11.9 [String utilities], page 178).

textual-fold kons knil textual :optional start end

textual-fold-right kons knil textual :optional start end

[Function]
[Function]
[R7RS text] {scheme.text} These are just aliases of srfi.13’s string-fold and string-fold-right, respectively (see Section 11.6.10 [SRFI-13 String mapping], page 699).

textual-map proc textual1 textual2 ...

[Function]
[R7RS text] {scheme.text}

textual-for-each proc textual1 textual2 ...

[Function]
[R7RS text] {scheme.text}

textual-map-index proc textual :optional start end

[Function]
[R7RS text] {scheme.text}

textual-for-each-index proc textual :optional start end

[Function]
[R7RS text] {scheme.text}

textual-count textual pred :optional start end

[Function]
[R7RS text] {scheme.text}

textual-filter pred textual :optional start end

[Function]
[R7RS text] {scheme.text}

textual-remove pred textual :optional start end

[Function]
[R7RS text] {scheme.text}

textual-replicate textual from to :optional start end

[Function]
[R7RS text] {scheme.text}

textual-split textual delimiter :optional grammar limit start end

[Function]
[R7RS text] {scheme.text} Like string-split, except that the returned strings are all immutable and indexed. See Section 6.11.9 [String utilities], page 178, for the details.

10.3.12 scheme.generator - R7RS generators

scheme.generator

[Module]
This module provides generators and accumulators. They were first defined in SRFI-121, then enhanced in SRFI-158, and finally incorporated R7RS large as (scheme generator).

A generator is a thunk to generate a sequence of values, potentially terminated by EOF. Procedures to deal with generators are provided by gauche.generator, which is a superset of SRFI-121. See Section 9.12 [Generators], page 436, for the details.

An accumulator is an opposite of generators. They are procedures that work as consumers. An accumulator takes one argument. When non-eof value is given, the value is stored, and when EOF is given, the accumulated value is returned. How the values are accumulated depends on the accumulator.

Once EOF is given, the accumulator is “finalized”. Subsequent EOF makes it return the same accumulated value. It is undefined if other values are passed after EOF is passed.

The accumulator can be used to parameterize procedures that yield aggregate objects. Consider the following procedure, which takes items from two generators and accumulate them alternatively. (Note that glet* is Gauche’s procedure but not in SRFI-158).

(define (intertwine acc gen1 gen2)
(let loop ()
  (glet* ([a (gen1)]
          [b (gen2)])
    (acc a)
    (acc b)
    (loop)))
(acc (eof-object)))

The procedure can return various types of collections, without knowing the actual type—the passed accumulator determines it.

(intertwine (list-accumulator) (giota 5) (giota 5 100))
⇒ (0 100 1 101 2 102 3 103 4 104)
(intertwine (vector-accumulator) (giota 5) (giota 5 100))
⇒ #(0 100 1 101 2 102 3 103 4 104)
(intertwine (bytevector-accumulator) (giota 5) (giota 5 100))
⇒ #u8(0 100 1 101 2 102 3 103 4 104)

Note: In Gauche, you can also use classes to parameterize returned container types (e.g. map-to), for many collection classes support builder protocol. See Section 9.6 [Collection framework], page 404, for the details. Accumulator has the flexibility that you can provide more than one ways to construct return value on the same type (e.g. forward and reverse list).

The following generator procedures are explained in {gauche}:section (see Section 9.12 [Generators], page 436):
• Section 9.12.1 [Generator constructors], page 436:
  generator circular-generator make-iota-generator
  make-range-generator make-coroutine-generator
  make-unfold-generator make-for-each-generator
  list->generator
  vector->generator reverse-vector->generator
  string->generator bytevector->generator
• Section 9.12.2 [Generator operations], page 440:
  gcons* gappend gflatten
  ggroup gmerge gmap
  gcombine gfilter gremove
  gstate-filter ggroup
  gtake gdrop gtake-while
  gdrop-while gdelete gdelete-neighbor-dups
  gindex gselect
• Section 9.12.3 [Generator consumers], page 445:
  generator->list generator->reverse-list generator->list
  generator->vector generator->vector! generator->string
  generator->count generator->any generator->every
  generator->unfold
• Section 6.15.11 [Folding generated values], page 231:
  generator-fold generator->for-each generator-find

The following are accumulator procedures:
make-accumulator kons knil finalizer
[Function]
[R7RS generator] {scheme.generator} Creates and returns an accumulator with a state, whose initial value is knil. When non-EOF value v is passed to the accumulator, kons is
called as \((\text{kons v state})\), and its result becomes the new state value. When EOF value is passed, \((\text{finalizer state})\) is called and its result becomes the result of accumulator.

\begin{verbatim}
list-accumulator [Function]
reverse-list-accumulator [Function]
\text{[R7RS generator] \{scheme.generator\} Creates and returns accumulators that return accumulated value as a list, in the accumulated order (list-accumulator) or the reverse order (reverse-list-accumulator).}

vector-accumulator [Function]
reverse-vector-accumulator [Function]
bytevector-accumulator [Function]
\text{[R7RS generator] \{scheme.generator\} Returns accumulators that return accumulated value as a fresh vector or bytevector (u8vector), in the accumulated order (vector-accumulator, bytevector-accumulator) or the reverse order (reverse-vector-accumulator). There's no reverse-bytevector-accumulator.}

vector-accumulator! vec at [Function]
bytevector-accumulator! bvec at [Function]
\text{[R7RS generator] \{scheme.generator\} The vec or bvec argument is a mutable vector or bytevector (u8vector), and is used as a buffer. Returns an accumulator that stores the accumulated values in the buffer, starting from the index at. It is an error if the accumulator gets more values after the buffer reaches at the end. Once EOF is passed to the accumulator, vec or bvec is returned, respectively.}

string-accumulator [Function]
\text{[R7RS generator] \{scheme.generator\} Returns an accumulator that accepts characters and accumulates them to a string.}

sum-accumulator [Function]
product-accumulator [Function]
count-accumulator [Function]
\text{[R7RS generator] \{scheme.generator\} Returns accumulators that yield a scalar value. The accumulator created by sum-accumulator and product-accumulator accepts numbers, and keep adding or multiplying it with the accumulated value (the default value is 0 and 1, respectively). The accumulator created by count-accumulator accepts any objects and just counting it.}

10.3.13 scheme.lseq - R7RS lazy sequences

scheme.lseq [Module]
This module provides lightweight lazy sequence (lseq), conceptually represented by a pair of element and generator. When the rest of sequence is taken, the generator is evaluated and yields another pair of element and generator, and so on. The overhead is one allocation of a pair per element. It is much lighter than streams (see Section 12.90 [Stream library], page 1033), which requires to create a thunk for every element.

Gauche already has built-in support for such lazy sequences; we go further to make it behave like ordinary pairs—that is, if you take cdr of a lazy pair, we automatically forces the generator so it is indistinguishable from an ordinary pair, modulo side effects. See Section 6.18.2 [Lazy sequences], page 241.

SRFI-127, the original srfi for this module, is a bit ambiguous whether its lazy sequence must be implemented with a pair whose cdr is a generator procedure, or it refers to the
pair+generator as a conceptual model. Considering of the purpose of lazy sequence, the concrete implementation shouldn’t matter; that is, the user of lazy sequence should not count on the fact that the lseq is an improper list terminated by a generator procedure. Instead, an lseq should be treated as an opaque object that can be passed to scheme.lseq procedures.

With that premise, we implement this module as just a thin wrapper of Gauche’s native lazy sequence. It is upper-compatible, except that the code that assumes the internal structure could break. Notably, the constructor generator->lseq is the same as Gauche’s built-in, which returns Gauche’s lseq, undistinguishable to the ordinary list.

```scheme
(procedure? (generator->lseq (generator 1)))
;; => #t, in SRFI-127 reference implementation,
;; #f, in our implementation.
```

### lseq? x

[R7RS lseq] {scheme.lseq} Returns true iff x is an object that can be passed to lseq procedures. In Gauche, it returns #t if x is a pair or an empty list, since a lazy pair is indistinguishable from a pair.

### lseq=? elt=? lseq1 lseq2

[R7RS lseq] {scheme.lseq} Compare two lseqs element-wise using elt=? and returns #t iff two lseqs are equal.

### lseq-car lseq

[R7RS lseq] {scheme.lseq} Returns the first item of lseq. If lseq is empty, an error is raised. In Gauche, these are just aliases of car.

### lseq-cdr lseq

[R7RS lseq] {scheme.lseq} Returns the rest of lseq. If lseq is empty, an error is raised. In Gauche, these are just aliases of cdr.

### lseq-take lseq k

[R7RS lseq] {scheme.lseq} Returns an lseq that has first k items, or an lseq that skips first k items, respectively.

An error is signaled when the resulting lseq of lseq-take reached at the end of sequence before k items are taken. It is different from Gauche’s `ltake`, which simply returns () in such case.

On the other hand, lseq-drop is the same as drop in Gauche; it just drops k items from the head of input sequence, regardless of whether it is an ordinary list or lseq.

### lseq-realize lseq

[R7RS lseq] {scheme.lseq} Realizes all the elements in lseq, resulting an ordinary list.

### lseq->generator lseq

[R7RS lseq] {scheme.lseq} Creates a generator from lseq. In Gauche, this is same as list->generator.

### lseq-length lseq

[R7RS lseq] {scheme.lseq} Returns the length of lseq. All the elements in lseq are realized as the side effect. In Gauche, this is same as length.
Chapter 10: Library modules - R7RS standard libraries

10.3.14 scheme.stream - R7RS stream

This module provides utilities for lazily evaluated streams. It is more heavyweight than lazy sequences (see Section 6.18.2 [Lazy sequences], page 241), but it strictly implements “as lazy as possible” semantics—elements are never evaluated until it is actually accessed.

The following procedures are provided in Gauche’s util.stream module; see Section 12.90 [Stream library], page 1033, for their description:

- stream-null
- stream-cons
- stream?
- stream-null?
- stream-pair?
- stream-car
- stream-cdr
- stream-lambda
- define-stream
- list->stream
- port->stream
- stream->list
- stream-append
- stream-concat
- stream-constant
- stream-drop-while
- stream-filter
- stream-fold
stream-for-each  stream-from  stream-iterate  stream-length
stream-let   stream-map  stream-match  stream-of
stream-range stream-ref  stream-reverse  stream-scan
stream-take-while stream-unfold stream-unfolds stream-zip

The following macro and procedures have different interface from Gauche’s util.stream module:

\texttt{stream} \hspace{1cm} \texttt{stream-take} \hspace{1cm} \texttt{stream-drop}

\begin{itemize}
  \item \texttt{stream \textit{expr} \ldots} \hspace{1cm} [Macro]
  \hspace{1cm} \{\texttt{scheme.stream}\} Returns a new stream whose elements are the result of \textit{expr} \ldots. Arguments won’t be evaluated until required.
  \item \texttt{stream-take \textit{n} \textit{stream}} \hspace{1cm} [Function]
  \hspace{1cm} \{\texttt{scheme.stream}\} Returns a stream that contains first \textit{n} elements from \textit{stream}, or elements without first \textit{n} elements from it, respectively. If \textit{stream} has less than \textit{n} elements, \texttt{stream-take} returns a copy of the entire \textit{stream}, while \texttt{stream-drop} returns a null stream.
\end{itemize}

10.3.15 scheme.box - R7RS boxes

\texttt{scheme.box} \hspace{1cm} [Module]

Gauche supports boxes built-in (see Section 6.17 [Boxes], page 238), so this module is merely a facade that exports the following identifiers:

\begin{itemize}
  \item \texttt{box} \hspace{1cm} \texttt{box?} \hspace{1cm} \texttt{unbox} \hspace{1cm} \texttt{set-box!}
\end{itemize}

10.3.16 scheme.list-queue - R7RS list queues

\texttt{scheme.list-queue} \hspace{1cm} [Module]

A library of simple queue based on lists. Gauche has a queue support in data.queue module, which also includes MT-safe queue (see Section 12.19 [Queue], page 837). This library is implemented on top of data.queue’s <queue> object and mainly provided for portable code.

The list-queue is just an instance of <queue>, so you can pass a queue created by make-queue to scheme.list-queue API and a list-queue created by make-list-queue to Gauche’s queue API.

Note: Some API of this library requires to return internal pairs the queue uses, for the efficiency. The pair’s car/cdr will be mutated by subsequent queue operation, and also any mutation done on the pair would cause inconsistency in the original queue.

\texttt{make-list-queue \textit{lis} :optional \textit{last}} \hspace{1cm} [Function]

\{\texttt{scheme.list-queue}\} Creates and returns a list-queue whose initial content is \textit{lis}. In Gauche, a list queue is just an instance of <queue> (see Section 12.19 [Queue], page 837).

The cells in \textit{lis} are owned by the queue; the caller shouldn’t mutate it afterwards, nor assume its structure remains the same.

The optional \texttt{last} argument must be the last pair of \textit{lis}. If it is passed, make-list-queue will skip scanning \textit{lis} and just hold a reference to \textit{last} as the tail of the queue.
**Chapter 10: Library modules - R7RS standard libraries**

### list-queue elt ...

[R7RS list-queue] {scheme.list-queue} Creates and returns a list-queue whose initial content is `elt ...`. In Gauche, a list queue is just an instance of `<queue>` (see Section 12.19 [Queue], page 837).

### list-queue-copy queue

[R7RS list-queue] {scheme.list-queue} Returns a copy of a list-queue `queue`.

### list-queue-unfold p f g seed :optional queue

[R7RS list-queue] {scheme.list-queue} Prepend `queue` with the items generated by `(unfold p f g seed)` and returns the updated queue. See Section 10.3.1 [R7RS lists], page 593, for `unfold`. If `queue` is omitted, a fresh queue is created.

```scheme
(list-queue-unfold (pa$ = 5) ; p
 (pa$ * 2) ; f
 (pa$ + 1) ; g
 0 ; seed
 (list-queue 'x 'y 'z))
⇒ a queue containing (0 2 4 6 8 x y z)
```

### list-queue-unfold-right p f g seed :optional queue

[R7RS list-queue] {scheme.list-queue} Append `queue` with the items generated by `(unfold-right p f g seed)` and returns the updated queue. See Section 10.3.1 [R7RS lists], page 593, for `unfold-right`. If `queue` is omitted, a fresh queue is created.

```scheme
(list-queue-unfold-right (pa$ = 5) ; p
 (pa$ * 2) ; f
 (pa$ + 1) ; g
 0 ; seed
 (list-queue 'x 'y 'z))
⇒ a queue containing (x y z 8 6 4 2 0)
```

### list-queue? obj

[R7RS list-queue] {scheme.list-queue} Returns true iff `queue` is a list-queue. In Gauche, it is the same as `queue?` in the `data.queue` module.

### list-queue-empty? queue

[R7RS list-queue] {scheme.list-queue} Returns true iff `queue` is empty. Same as `queue-empty?` of `data.queue`.

### list-queue-front queue

[R7RS list-queue] {scheme.list-queue} Returns the front element of the `queue`. An error is thrown if `queue` is empty. Same as `queue-front` of `data.queue`.

### list-queue-back queue

[R7RS list-queue] {scheme.list-queue} Returns the rear element of the `queue`. An error is thrown if `queue` is empty. Same as `queue-rear` of `data.queue`.

### list-queue-list queue

[R7RS list-queue] {scheme.list-queue} Returns the internal list of `queue`. Note that the list would be modified by subsequent operations of `queue`, and any modification on the list would make `queue` inconsistent. The primary purpose of this procedure is to implement other queue-related operations with small overhead.

If you merely need a cheap access the content of the queue, consider `list-queue-remove-all!`. That returns the list of elements of the queue without copying, and simultaneously reset the queue to empty, so it’s safe.
Chapter 10: Library modules - R7RS standard libraries

list-queue-first-last queue
[Function]
[R7RS list-queue] {scheme.list-queue} Returns two values, the first and last pair of queue. If the queue is empty, two empty lists are returned. This also returns the internal pair of the queue, so any subsequent operations of queue would change the contents of the pairs, and any modification on the pairs would make queue inconsistent. The purpose of this procedure is to implement other queue-related operations with small overhead. This procedure should not be used in general.

list-queue-add-front! queue elt
[Function]
[R7RS list-queue] {scheme.list-queue} Add elt to the front of queue. Same as (queue-push! queue elt) of data.queue.

list-queue-add-back! queue elt
[Function]
[R7RS list-queue] {scheme.list-queue} Add elt to the back of queue. Same as (enqueue! queue elt) of data.queue.

list-queue-remove-front! queue
[Function]
[R7RS list-queue] {scheme.list-queue} Remove an element from the front of queue and returns the removed element. Throws an error if queue is empty. Same as dequeue! of data.queue.

list-queue-remove-back! queue
[Function]
[R7RS list-queue] {scheme.list-queue} Remove an element from the back of queue and returns the removed element. Throws an error if queue is empty. This isn’t guaranteed to be efficient; it is O(n) operation where n is the number of elements. In general, if you need this operation frequently, you should consider double-ended queue. (See Section 12.16 [Immutable deques], page 833, and also see Section 12.22 [Ring buffer], page 849.)

list-queue-remove-all! queue
[Function]
[R7RS list-queue] {scheme.list-queue} Remove all the elements from queue and returns them as a list. The list isn’t copied—this is O(1) operation. This should be preferred over list-queue-list, for it’s safer. In Gauche, this is the same as dequeue-all! in data.queue.

list-queue-set-list! queue lis :optional last
[Function]
[R7RS list-queue] {scheme.list-queue} Modify queue to have the elements in lis as its element. The original content of queue is discarded. If the optional last argument is provided, it must be the last pair of lis, and the procedure uses that instead of scanning lis, to achieve O(1) operation.

list-queue-append queue ...
[Function]
[R7RS list-queue] {scheme.list-queue} Returns a fresh list-queue whose contents are concatenation of queues. The contents of arguments are intact. This is O(n) operation where n is the total number of elements.

list-queue-append! queue ...
[Function]
[R7RS list-queue] {scheme.list-queue} Returns a list-queue whose contents are concatenation of queues. During the operation, the contents of queues may be mutated, and they shouldn’t be used any longer. (In Gauche, to avoid accident, we actually empty all the queues.) It is also noted that the result doesn’t need to be eq? to any of the arguments. This is O(m) operation where m is the total number of queues (as opposed to the number of elements).
list-queue-concatenate queues

[Function]
[R7RS list-queue] {scheme.list-queue} (apply list-queue-append queues).

list-queue-map proc queue

[Function]
[R7RS list-queue] {scheme.list-queue} Returns a fresh list-queue whose elements are obtained by applying proc on every elements in queue.

list-queue-map! proc queue

[Function]
[R7RS list-queue] {scheme.list-queue} Replaces every element in queue by the result of application of proc on the element.

list-queue-for-each proc queue

[Function]
[R7RS list-queue] {scheme.list-queue} Applies proc on every element of queue. The results are discarded.

10.3.17 scheme.ephemeron - R7RS ephemeron

scheme.ephemeron

[Module]

This module defined ephemeron, a weak reference structure to hold key-value association. This is originally defined as SRFI-142.

Gauche supports weak pointers in the form of weak vectors (see Section 6.13.4 [Weak vectors], page 205), but it is known that a simple weak pointer (a single pointer that doesn’t prevent the pointed object from being collected) isn’t enough to implement weak key-value association such as mappings.

An ephemeron is a record that points to a key and an associated datum, with the following characteristics:

1. Reference to the key is weak; it doesn’t prevent the key from being collected if there’s no strong reference to the key, except from the datum associated by the ephemeron.

2. Reference to the datum is also a kind of weak; it doesn’t prevent the datum from being collected if there’s no strong reference to the datum, and there’s no strong reference to the associated key. Note that the datum is retained as long as the key is retained, even there’s no strong reference to the datum itself.

Implementing the proper ephemeron requires deep integration with the GC. At this moment, Gauche’s ephemeron is implemented separately from GC, and has the following limitations:

• If the datum has a strong reference to the associated key, the key won’t be collected even if there’s no other strong reference to it.

• After the key is collected and there’s no strong reference to the datum, ephemeron-broken? needs to be called in order to trigger the collection of the datum.

Since the timing of collection isn’t specified in the spec, Gauche’s implementation still conforms SRFI-142, but in practice you need to be aware of these limitations. Eventually we want to support full ephemeron integrated with GC.

Once the key and/or the datum is collected (we call such ephemeron “broken”), referencing them returns a bogus value. The proper way to use an ephemeron e is the following pattern:

(let ([k (ephemeron-key e)])
  [d (ephemeron-datum e)])
(if (ephemeron-broken? e)
  (... k and d are invalid ....)
  (... k and d are valid ....)))

You should take values, then check if the ephemeron isn’t broken yet. If you call ephemeron-broken? first, there’s a chance that the ephemeron is broken between the check and the time you reference it.
Chapter 10: Library modules - R7RS standard libraries

make-ephemeron  key  datum  
[R7RS ephemeron]  {scheme.ephemeron}  Create a new ephemeron associating the key to the datum.

ephemeron?  obj  
[R7RS ephemeron]  {scheme.ephemeron}  Returns #t iff obj is an ephemeron.

ephemeron-key  ephemeron  
ephemeron-datum  ephemeron  
[R7RS ephemeron]  {scheme.ephemeron}  Returns the key and the datum of ephemeron, respectively. If the ephemeron is already broken, there’s no guarantee on what is returned. Thus you should always call ephemeron-broken? after calling these procedure to ensure the values are meaningful. See the scheme.ephemeron entry for the details.

ephemeron-broken?  ephemeron  
[R7RS ephemeron]  {scheme.ephemeron}  Returns #t iff ephemeron has been broken, that is, its key and/or datum may be collected and cannot be reliably retrieved. See the scheme.ephemeron entry for the details.

reference-barrier  key  
[R7RS ephemeron]  {scheme.ephemeron}  This procedure does nothing by itself, but guarantees key is strongly reference until returning from this procedure.

10.3.18 scheme.comparator - R7RS comparators

scheme.comparator  
[Module]  
This module defines comparators and related procedures. Originally called srfi.128.

Gauche supports comparators fully compatible to scheme.comparator built-in. See Section 6.2.4 [Basic comparators], page 116, for the following procedures defined in this module.

comparator?  comparator-ordered?  comparator-hashable?  
make-comparator  make-pair-comparator  
make-list-comparator  make-vector-comparator  
make-eq-comparator  make-eqv-comparator  make-equal-comparator  

boolean-hash  char-hash  char-ci-hash  string-hash  
string-ci-hash  symbol-hash  number-hash  
hash-bound  hash-salt  

make-default-comparator  default-hash  
comparator-register-default!  

comparator-type-test-predicate  comparator-equality-predicate  
comparator-ordering-predicate  comparator-hash-function  
comparator-test-type  comparator-check-type  comparator-hash  

=?  <=?  >=?  comparator-if<=>

10.3.19 scheme.regex - R7RS regular expressions

scheme.regex  
[Module]  
This module provides operations on Scheme Regular Expressions (SRE). Originally defined as srfi.115.
Gauche has built-in support of regular expressions, and this module simply translates SRE to Gauche’s native regular expressions. The regular expression object returned from the `regexp` procedure in this module is Gauche’s `<regexp>` object, for example. You can pass Gauche’s regexp object to the procedures that expects compiled SRE in this module as well.

### Scheme regular expression syntax

#### Syntax summary

SRE is just an S-expression with the structure summarized below.

With the exception of or, any syntax that takes multiple `<sre>` processes them in a sequence. In other words `(foo <sre> ...)` is equivalent to `(foo (seq <sre> ...))`.

Note: SRE uses the symbol `|` for alteration, but the vertical bar character is used for symbol escape in Gauche (and R7RS), so you have to write such symbol as `\|`. We recommend to use `or` instead.

```
<sre> ::=  
| <string> ; A literal string match. 
| <cset-sre> ; A character set match. 
| (* <sre> ...) ; 0 or more matches. 
| (+ <sre> ...) ; 1 or more matches. 
| (one-or-more <sre> ...) ; 0 or 1 matches. 
| (optional <sre> ...) ; <n> matches. 
| (exactly <n> <sre> ...) ; <n> or more matches. 
| (at-least <n> <sre> ...) ; > <n> matches. 
| (** <n> <m> <sre> ...) ; <n> to <m> matches. 
| (repeated <n> <m> <sre> ...) 
| (\|\| <sre> ...) ; Alternation. 
| (or <sre> ...) 
| (: <sre> ...) ; Sequence. 
| (seq <sre> ...) 
| ($) <sre> ...) ; Numbered submatch. 
| (submatch <sre> ...) ; Named submatch. <name> is 
| (submatch-named <name> <sre> ...) ; a symbol. 
| (w/case <sre> ...) ; Introduce a case-sensitive context. 
| (w/nocase <sre> ...) ; Introduce a case-insensitive context. 
| (w/unicode <sre> ...) ; Introduce an unicode context. 
| (w/ascii <sre> ...) ; Introduce an ascii context. 
| (w/nocapture <sre> ...) ; Ignore all enclosed submatches. 
| bos ; Beginning of string. 
| eos ; End of string. 
| bol ; Beginning of line. 
| eol ; End of line. 
| bow ; Beginning of word. 
| eow ; End of word. 
| nwb ; A non-word boundary. 
| word ; A single word. 
```
Chapter 10: Library modules - R7RS standard libraries

The grammar for `cset-sre` is as follows.

```
<range-spec> ::= <string> | <char>
```

Basic patterns

```
<string>
A literal string.

(regexp-search "needle" "hayneedlehay")
⇒ #<regexp-match>
(regexp-search "needle" "haynEEdlehay")
⇒ #f
```

```
(seq <sre> ...)
(: <sre> ...)
A sequence of patterns that should be matched in the same order. This is the same as RE syntax (?:re...)
```
(regexp-search '(: "one" space "two" space "three") "one two three")
⇒ #<regexp-match>
(or <sre> ...)
(\|<sre> ...)
Matches one of the given patterns. This is the same as RE syntax pattern1|pattern2|...
(regexp-search '((or "eeney" "meeney" "miney") "meeney")
⇒ #<regexp-match>
(regexp-search '((or "eeney" "meeney" "miney") "moe")
⇒ #f
(w/nocase <sre> ...)
Changes to match the given patterns case-insensitively. Sub-patterns can still be made sensitive with w/case. This is the same as RE syntax (?i:re...)
(regexp-search "needle" "haynEEdlehay") ⇒ #f
(regexp-search '(w/nocase "needle") "haynEEdlehay")
⇒ #<regexp-match>
(regexp-search '(~ (w/nocase ("Aab"))) "B") ⇒ #f
(regexp-search '(~ (w/nocase ("Aab"))) "b") ⇒ #f
(w/case <sre> ...)
Changes to match the given patterns case-sensitively. Sub-patterns can still be made case-insensitive. This is the same as RE syntax (?-i:re...). This is the default.
(regexp-search '(w/nocase "SMALL" (w/case "BIG")) "smallBIGsmall")
⇒ #<regexp-match>
(regexp-search '(w/nocase (~ (w/case ("Aab"))) "b") ⇒ #f
(w/ascii <sre> ...)
Limits the character sets and other predefined patterns to ASCII. This affects patterns or character sets like any, alpha, (word)... (regexp-search '(w/ascii bos (* alpha) eos) "English")
⇒ #<regexp-match>
(regexp-search '(w/ascii bos (* alpha) eos) "Ελληνική") ⇒ #f
(w/unicode <sre> ...)
Changes the character sets and other predefined patterns back to Unicode if w/ascii has been used in the outer scope. This is the default.
(regexp-search '(w/unicode bos (* alpha) eos) "English")
⇒ #<regexp-match>
(regexp-search '(w/unicode bos (* alpha) eos) "Ελληνική")
⇒ #<regexp-match>
(w/nocapture <sre> ...)
Disables capturing for all submatch and submatch-named inside.
(let ((number '(($ (+ digit)))))
  (cdr
    (regexp-match->list
      (regexp-search '(: ,number "-" ,number "-" ,number)
(cdr
  (regexp-match->list
   (regexp-search '(: ,number "-" (w/nocapture ,number) "-" ,number)
   "555-867-5309")
   "555-867-5309"))) ⇒ '("555" "867" "5309")

Repeating patterns

(optional <sre> ...)

(?: <sre> ...)

Matches the pattern(s) one or zero times.

(regexp-search '(: "match" (? "es") "!") "matches!")
⇒ #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "match!")
⇒ #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "match!")
⇒ #f

(zero-or-more <sre> ...)

(\* <sre> ...)

Matches the pattern(s) zero or more times.

(regexp-search '(: "<" (\* (~ #\>) ) ">") "<html>")
⇒ #<regexp-match>
(regexp-search '(: "<" (\* (~ #\>) ) ">") "">
⇒ #<regexp-match>
(regexp-search '(: "<" (\* (~ #\>) ) ">") "<html>")
⇒ #f

(one-or-more <sre> ...)

(+ <sre> ...)

Matches the pattern(s) at least once.

(regexp-search '(: "<" (\+ (~ #\>) ) ">") "<html>")
⇒ #<regexp-match>
(regexp-search '(: "<" (\+ (~ #\>) ) ">") "<a>")
⇒ #<regexp-match>
(regexp-search '(: "<" (\+ (~ #\>) ) ">") "">
⇒ #f

(at-least n <sre> ...)

(\= n <sre> ...)

Matches the pattern(s) at least n times.

(regexp-search '(: "<" (\= 3 (~ #\>) ) ">") "<table>")
⇒ #<regexp-match>
(regexp-search '(: "<" (\= 3 (~ #\>) ) ">") "<pre>")
⇒ #<regexp-match>
(regexp-search '(: "<" (\= 3 (~ #\>) ) ">") "<tr>")
⇒ #f

(exactly n <sre> ...)

(\= n <sre> ...)

Matches the pattern(s) exactly n times.

(regexp-search '(: "<" (\= 4 (~ #\>) ) ">") "<html>")
⇒ #<regexp-match>
(regexp-search '(: "<" (\= 4 (~ #\>) ) ">") "<table>")
⇒ #f

(repeated from to <sre> ...)
(** from to <sre> ...)

Matches the pattern(s) at least from times and up to to times.

(regexp-search ': ( = 3 (** 1 3 numeric) ".") (** 1 3 numeric))
"192.168.1.10")
⇒ #<regexp-match>
(regexp-search ': ( = 3 (** 1 3 numeric) ".") (** 1 3 numeric))
"192.0168.1.10")
⇒ #f

Submatch Patterns

(submatch <sre> ...)
($ <sre> ...)

Captures the matched string. Each capture is numbered increasing from one (cap-
ture zero is the entire matched string). For nested captures, the numbering scheme
is depth-first walk.

(submatch-named <name> <sre> ...)
(-> <name> <sre> ...)

Captures the matched string and assigns a name to it in addition to a number. This
is the equivalent of (?<name>re...)

(backref <n-or-name>)

Matches a previously matched submatch. This is the same as RE syntax \n or \k<name>.

Character Sets

<char>

A character set contains a single character.

(regexp-matches '(* #\-) "---") ⇒ #<regexp-match>
(regexp-matches '(* #\-) "_-") ⇒ #f

"<char>

A character set contains a single character. This is technically ambiguous with SRE
matching a literal string. However the end result of both syntaxes is the same.

<char-set>

A SRFI-14 character set.

Note that while currently there is no portable written representation of SRFI
14 character sets, you can use Gauche reader syntax #\[char-set-spec], see
Section 6.10 [Character sets], page 165.

(regexp-partition '(+ ,char-set:vowels) "vowels")
⇒ ("v" "o" "w" "e" "1s")

(char-set <string>)

A character set contains the characters in the given string. This is the same as
'(char-set ,(<string->char-set <string>)));

(regexp-matches '(* ("aeiou")) "oui") ⇒ #<regexp-match>
(regexp-matches '(* ("aeiou")) "ouais") ⇒ #f
(regexp-matches '(* ("e\x0301")) "e\x0301") ⇒ #<regexp-match>
(regexp-matches '("e\x0301") "e\x0301") ⇒ #f
(regexp-matches '("e\x0301") "e") ⇒ #<regexp-match>
(regexp-matches '("e\x0301") "\x0301") ⇒ #<regexp-match>
(regexp-matches '("e\x0301") "\x00E9") ⇒ #f

(char-range <range-spec> ...)
(/ <range-spec> ...)
A character set contains the characters within <range-set>. This is the same as RE syntax [].
(regexp-matches '(* (/ "AZ09") "R2D2") ⇒ #<regexp-match>
(regexp-matches '(* (/ "AZ09") "C-3PO") ⇒ #f

(or <cset-sre> ...)
(|| <cset-sre> ...)
A shorthand for '(char-set ,char-set-union <cset-sre>...).

(complement <cset-sre> ...)
(~ <cset-sre> ...)
A shorthand for '(char-set ,char-set-complement <cset-sre>...).

(difference <cset-sre> ...)
(- <cset-sre> ...)
A shorthand for '(char-set ,char-set-difference <cset-sre>...).
(regexp-matches '(* (- (/ "az") ("aeiou"))) "xyzzy")
⇒ #<regexp-match>
(regexp-matches '(* (- (/ "az") ("aeiou"))) "vowels")
⇒ #f

(and <cset-sre> ...)
(& <cset-sre> ...)
A shorthand for '(char-set ,char-set-intersection <cset-sre>...).
(regexp-matches '(* (& (/ "az") (~ ("aeiou"))) "xyzzy")
⇒ #<regexp-match>
(regexp-matches '(* (& (/ "az") (~ ("aeiou"))) "vowels")
⇒ #f

(w/case <cset-sre>)
(w/nocase <cset-sre>)
(w/ascii <cset-sre>)
(w/unicode <cset-sre>)
This is similar to the SRE equivalent, listed to indicate that they can also be applied on character sets.

Named Character Sets
Note that if w/ascii is in effect, these character sets will return the ASCII subset. Otherwise they return full Unicode ones.

any
Matches any character. This is the . in regular expression.

nonl
Matches any character other than \return or \newline.

ascii
lower-case
lower
A shorthand for (char-set:lower-case).

upper-case
upper

title-case
title
A shorthand for (char-set:title-case).

alphabetic
alpha

numeric
num
A shorthand for (char-set:digit).

alphanumeric
alphanum
alnum

punctuation
punct
A shorthand for (char-set:punctuation).

symbol
A shorthand for (char-set:symbol).

graphic
graph
A shorthand for (char-set:graphic).
(or alphanumeric punctuation symbol)

whitespace
white
space
A shorthand for (char-set:whitespace).

printing
print
A shorthand for (char-set:printing).

control
cntrl
A character set contains ASCII characters with from 0 to 31.

hex-digit
xdigit
A shorthand for (char-set:hex-digit).
Boundary Assertions

bos
eos

Matches the beginning of the string. If start/end parameters are specified, matches the start or end of the substring as specified.

bol
eol

Matches the beginning or end of a line (or the string). For single line matching, this is the same as bos and eos. A line is interpreted the same way with read-line.

bow
eow

Matches the beginning or the end of a word.

(bypass-regexp-search : (bow "foo") "foo") ⇒ #<regexp-match>
(bypass-regexp-search : (bow "foo") "<foo>"") ⇒ #<regexp-match>
(bypass-regexp-search : (bow "foo") "snafoo") ⇒ #f
(bypass-regexp-search : "foo" eow) "foo") ⇒ #<regexp-match>
(bypass-regexp-search : "foo" eow) "foo!" ) ⇒ #<regexp-match>
(bypass-regexp-search : "foo" eow) "foobar") ⇒ #f

nwb

A shorthand for (neg-look-ahead (or bow eow)).

(word <sre> ...)  

Matches the word boundary around the given SRE:

(: bow <sre> ... eow)

(word+ <cset-sre> ...)  

Matches a single word composed of characters of the given characters sets:

(word (+ (and (or alphanumeric "_") (or cset-sre ...))))

word

A shorthand for (word+ any).

bog
eog

Matches the beginning or end of a grapheme cluster. See Section 9.37.2 [Unicode text segmentation], page 553, for the low-level grapheme cluster segmentation.

grapheme

Matches a single grapheme cluster. See Section 9.37.2 [Unicode text segmentation], page 553, for the low-level grapheme cluster segmentation.

Non-Greedy Patterns

(non-greedy-optional <sre> ...)  

(?? <sre> ...)  

The non-greedy equivalent of (optional <sre>...). This is the same as RE syntax re??

(non-greedy-zero-or-more <sre> ...)  

(*/? <sre> ...)  

The non-greedy equivalent of (zero-or-more <sre>...). This is the same as RE syntax re*?
(non-greedy-repeated <m> <n> <sre> ...)
(??? <m> <n> <sre> ...)
   The non-greedy equivalent of (repeated <sre>...). This is the same as RE syntax re{n,m}?

(atomic <sre> ...)
   Atomic clustering. Once <sre> ... matches, the match is fixed; even if the following pattern fails, the engine won’t backtrack to try the alternative match in <sre> ....
   This is Gauche extension and is the same as RE syntax (?>pattern)

Look Around Patterns

(look-ahead <sre> ...)
   Zero-width look-ahead assertion. Asserts the sequence matches from the current position, without advancing the position. This is the same as RE syntax (?=pattern)
   (regexp-matches '( (: "regular" (look-ahead " expression") " expression")
                     "regular expression")
   => #<regexp-match>
   (regexp-matches '( (: "regular" (look-ahead")"expression")
                     "regular expression")
   => #f

(look-behind <sre> ...)
   Zero-width look-behind assertion. Asserts the sequence matches behind the current position, without advancing the position. It is an error if the sequence does not have a fixed length. This is the same as RE syntax (?<pattern)

(neg-look-ahead <sre> ...)
   Zero-width negative look-ahead assertion. This is the same as RE syntax (!=pattern)

(neg-look-behind <sre> ...)
   Zero-width negative look-behind assertion. This is the same as RE syntax (?<!pattern)

Using regular expressions

regexp re
   [Function]
   [R7RS regex] {scheme.regex} Compiles the given Scheme Regular Expression into a <regexp> object. If re is already a regexp object, the object is returned as-is.

rx sre ...
   [Macro]
   [R7RS regex] {scheme.regex} A macro shorthand for (regexp `(sre ...)).

regexp->sre re
   [Function]
   [R7RS regex] {scheme.regex} Returns the SRE corresponding to the given given regexp object. Note that if the regexp object is not created from an SRE, it may contain features that cannot be expressed in SRE and cause an error.

char-set->sre char-set
   [Function]
   [R7RS regex] {scheme.regex} Returns the SRE of the given character set. Currently this is not optimized. If you convert any to SRE for example, you may get an SRE listing every single character.

valid-sre? obj
   [Function]
   [R7RS regex] {scheme.regex} Returns true iff obj can be safely passed to regexp.
regexp? obj  
    [Function]  
    [R7RS regex] {scheme.regex} Returns true iff obj is a regexp.

regexp-matches re str [start [end]]  
    [Function]  
    [R7RS regex] {scheme.regex} Returns an <regexp-match> object if re successfully matches the entire string str or optionally from start (inclusive) to end (exclusive), or #f is the match fails.

For convenience, end accepts #f and interprets it as the end of the string.

The regexp-match object will contain information needed to extract any submatches.

regexp-matches? re str [start [end]]  
    [Function]  
    [R7RS regex] {scheme.regex} Similar to regexp-matches but returns #t instead of a <regexp-match> object.

regexp-search re str [start [end]]  
    [Function]  
    [R7RS regex] {scheme.regex} Similar to regexp-matches except that re only has to match a substring in str instead.

regexp-fold re kons knil str [finish [start [end]]]  
    [Function]  
    [R7RS regex] {scheme.regex} Calls the procedure kons for every match found in str with following four arguments:

    * The position of the end of the last matched string.
    * The <regexp-match> object.
    * The argument str.
    * The result of the last kons call or knil if this is the first call.

If finish is given, it is called after all matches with the same parameters as calling kons except that #f is passed instead of <regexp-match> and the result is returned. Otherwise the result of the last kons call is returned.

(regexp-fold 'word  
    (lambda (i m str acc)  
        (let ((s (regexp-match-submatch m 0)))  
            (cond ((assoc s acc)  
                => (lambda (x) (set-cdr! x (+ 1 (cdr x))) acc))  
                  (else `(((,s . 1) ,@acc)))))  
        '())  
    "to be or not to be")  
⇒ '(("not" . 1) ("or" . 1) ("be" . 2) ("to" . 2))

regexp-extract re str [start [end]]  
    [Function]  
    [R7RS regex] {scheme.regex} Returns a list of matched string or an empty list if no matches.

(regexp-extract '(+ numeric) "192.168.0.1")  
⇒ ("192" "168" "0" "1")

regexp-split re str [start [end]]  
    [Function]  
    [R7RS regex] {scheme.regex} Returns a list of not matched substrings. This can be seen as the opposite of regexp-extract where the matched strings are removed instead of returned.

(regexp-split '(+ space) "fee fi fo\tfum\n")  
⇒ ("fee" "fi" "fo" "fum")
(regexp-split '(';") "a,\,b,")  
⇒ ("a" "b" "")
(regexp-split '(* numeric) "abc123def456ghi789")  
⇒ ("abc" "def" "ghi" "")
regexp-partition re str [start [end]]

[Function]
[R7RS regex] {scheme.regex} Returns a list of all matched and not matched substrings. In
other words it’s the combination of regexp-extract and regexp-split where the boundary
of matched strings are used to split the original string.

\[
\begin{align*}
(\text{regexp-partition } '(+ (or space punct)) "") & \Rightarrow "" \\
(\text{regexp-partition } '(+ (or space punct)) "Hello, world!\n") & \Rightarrow ("Hello" "," "world" "\n") \\
(\text{regexp-partition } '(+ (or space punct)) "¿Dónde Estás?"") & \Rightarrow ("" "," "Dónde" "" "Estás" "?") \\
(\text{regexp-partition } '(* numeric) "abc123def456ghi789") & \Rightarrow ("abc" "123" "def" "456" "ghi" "789")
\end{align*}
\]

regexp-replace re str subst [start [end [count]]]

[Function]
[R7RS regex] {scheme.regex} Returns a new string where the first matched substring is
replaced with subst. If count is specified, the count-th match will be replaced instead of the
first one.

subst can be either a string (the replacement), an integer or a symbol to refer to the capture
group that will be used as the replacement, or a list of those.

The special symbols pre and post use the substring to the left or right of the match as
replacement, respectively.

subst could also be a procedure, which is called with the given match object and the result
will be used as the replacement.

The optional parameters start and end essentially transform the substitution into this

\[
(\text{regexp-replace re (substring str start end) subst})
\]

except that end can take #f which is the same as (string-length str).

\[
\begin{align*}
(\text{regexp-replace } '(+ space) "one two three" ",") & \Rightarrow "one\_two\_three" \\
(\text{regexp-replace } '(+ space) "one two three" "," 1 10) & \Rightarrow "ne\_two\ th" \\
(\text{regexp-replace } '(+ space) "one two three" "," 0 #f 0) & \Rightarrow "one\_two\ three" \\
(\text{regexp-replace } '(+ space) "one two three" "," 0 #f 1) & \Rightarrow "one\_two\_three" \\
(\text{regexp-replace } '(+ space) "one two three" "," 0 #f 2) & \Rightarrow "one\_two\ three"
\end{align*}
\]

Note that Gauche also has a builtin procedure of the same name, but works slightly differently,
see Section 6.12.2 [Using regular expressions], page 186.

regexp-replace-all re str subst [start [end]]

[Function]
[R7RS regex] {scheme.regex} Returns a new string where all matches in str are replaced
with subst. subst can also take a string, a number, a symbol or a procedure similar to
regexp-replace.

\[
\begin{align*}
(\text{regexp-replace-all } '(+ space) "one two three" ",") & \Rightarrow "one\_two\_three"
\end{align*}
\]

Note that Gauche also has a builtin procedure of the same name, but works slightly differently,
see Section 6.12.2 [Using regular expressions], page 186.

regexp-match? obj

[Function]
[R7RS regex] {scheme.regex} Returns true iff obj is a <regexp-match> object.

\[
(\text{regexp-match? } (\text{regexp-matches } "x" "x")) \Rightarrow \#t
\]
(regexp-match? (regexp-matches "x" "y")) ⇒ #f

regexp-match-count regexp-match  
[R7RS regex] {scheme.regex} Returns the number of matches in match except the implicit zero full match. This is just an alias of rxmatch-num-matches minus one.

(regexp-match-count (regexp-matches "x" "x")) ⇒ 0
(regexp-match-count (regexp-matches '($ "x") "x")) ⇒ 1

regexp-match-submatch regexp-match field  
[R7RS regex] {scheme.regex} This is an alias of regexp-match-substring

(regexp-match-submatch (regexp-search 'word "**foo**") 0) ⇒ "foo"
(regexp-match-submatch (regexp-search '(: "*" ($ word) "*" "**foo**") 0) ⇒ "*foo*"
(regexp-match-submatch (regexp-search '(: "*" ($ word) "*" "**foo**") 1) ⇒ "foo"

regexp-match-submatch-start regexp-match field  
[R7RS regex] {scheme.regex} This is an alias of regexp-match-submatch-start.

(regexp-match-submatch-start (regexp-search 'word "**foo**") 0) ⇒ 2
(regexp-match-submatch-start (regexp-search '(: "*" ($ word) "*" "**foo**") 0) ⇒ 1
(regexp-match-submatch-start (regexp-search '(: "*" ($ word) "*" "**foo**") 1) ⇒ 2

regexp-match-submatch-end regexp-match field  
[R7RS regex] {scheme.regex} This is an alias of regexp-match-submatch-end.

(regexp-match-submatch-end (regexp-search 'word "**foo**") 0) ⇒ 5
(regexp-match-submatch-end (regexp-search '(: "*" ($ word) "*" "**foo**") 0) ⇒ 6
(regexp-match-submatch-end (regexp-search '(: "*" ($ word) "*" "**foo**") 1) ⇒ 5

regexp-match->list regexp-match  
[R7RS regex] {scheme.regex} This is an alias of rxmatch-substrings

(regexp-match->list (regexp-search '(: ($ word) (+ (or space punct)) ($ word)) "cats & dogs"))⇒ '("cats & dogs" "cats" "dogs")

10.3.20 scheme.mapping - R7RS mappings

scheme.mapping  
[Module]

scheme.mapping.hash  
[Module]

This module defines immutable mappings from keys to values. Originally called srfi.146 and srfi.146.hash.

The scheme.mapping module provides mapping objects, where keys have total order. The scheme.mapping.hash module provides hashmap objects, where keys can be hashed.

Currently, Gauche uses built-in <tree-map> for the mapping object (see Section 6.14.2 [Treemaps], page 212), and built-in <hash-table> for the hashmap object (see Section 6.14.1 [Hashtables], page 206). The actual implementation may be changed in future versions, so the user must not rely on the underlying implementations.
The caller must treat mappings and hashmaps as immutable object. The modules also provide “linear update” APIs, which is allowed to mutate the mappings passed to the arguments, under assumption that the argument won’t be used afterwards. The linear update APIs are marked with ! at the end of the name. You should always use the returned value of the linear update APIs, for the side effect isn’t guaranteed.

10.3.20.1 Mappings

<mapping> [Class]
{scheme.mapping} The class for the mappings. On Gauche, this is just an alias of <tree-map>.

Constructors

mapping comparator key value . . . [Function]
[R7RS mapping] {scheme.mapping} Creates a new mapping with the given comparator, whose initial content is provided by key value . . . .

The comparator argument must be a comparator, with comparison/ordering procedure (see Section 6.2.4 [Basic comparators], page 116).

The key value . . . arguments must be even length, alternating keys and values.

    (define m (mapping default-comparator 'a 1 'b 2))

    (mapping-ref m 'a) ⇒ 1
    (mapping-ref m 'b) ⇒ 2

mapping-unfold p f g seed comparator [Function]
[R7RS mapping] {scheme.mapping} Creates a new mapping, whose content is populated by three procedures, p, f and g, and a seed value seed, as follows.

In each iteration, we have a current seed value, whose initial value is seed.

First, p, a stop predicate, is applied to the current seed value. If it returns true, we stop iteration and returns the new mapping.

Next, f is applied to the current seed value. It must return two values. The first one is for a key and the second one for the value. We add this pair to the mapping.

Then, g is applied to the current seed value. The result becomes the seed value of the next iteration. And we iterate.

The following example creates a mapping that maps ASCII characters to their character codes:

    (mapping-unfold (cut >= <> 128)
        (^c (values (integer->char c) c))
        (cut + <> 1)
        0
        default-comparator)

mapping/ordered comparator key value . . . [Function]
[R7RS mapping] {scheme.mapping} Similar to mapping, but keys are given in the ascending order w.r.t. the comparator. An implementation may use more efficient algorithm than mapping. In Gauche, this is the same as mapping at this moment.

mapping-unfold/ordered p f g seed comparator [Function]
[R7RS mapping] {scheme.mapping} Similar to mapping-unfold, but keys are generated in the ascending order w.r.t. the comparator. An implementation may use more efficient algorithm than mapping-unfold. In Gauche, this is the same as mapping-unfold at this moment.
Predicates

mapping? obj
[Function]  
[R7RS mapping]  
{scheme.mapping} Returns #t iff obj is a mapping object.

mapping-empty? m
[Function]  
[R7RS mapping]  
{scheme.mapping}  
M must be a mapping. Returns #t if m is empty, #f otherwise. In Gauche, this is same as tree-map-empty? (see Section 6.14.2 [Treemaps], page 212).

mapping-contains? m key
[Function]  
[R7RS mapping]  
{scheme.mapping}  
M must be a mapping. Returns #t if m has an entry with key, #f otherwise. In Gauche, this is same as tree-map-exists? (see Section 6.14.2 [Treemaps], page 212).

mapping-disjoint? m1 m2
[Function]  
[R7RS mapping]  
{scheme.mapping}  
Returns #t iff two mappings m1 and m2 have no keys in common. In other words, there’s no such key K that satisfy both (mapping-contains? m1 K) and (mapping-contains? m2 K).

Accessors

mapping-ref m key :optional failure success
[Function]  
[R7RS mapping]  
{scheme.mapping}  
Get the value from a mapping m associated with key, and calls success on the value, and returns its result. If m doesn’t have key, failure is invoked with no arguments and its result is returned. Both success and failure is called in tail context. When failure is omitted and key is not found, an error is signaled. When success is omitted, identity is assumed.

mapping-ref/default m key default
[Function]  
[R7RS mapping]  
{scheme.mapping}  
Returns the value associated to key from a mapping m. If m doesn’t have key, default is returned.

mapping-key-comparator m
[Function]  
[R7RS mapping]  
{scheme.mapping}  
Returns a comparator used to compare keys in a mapping m. See Section 6.2.4 [Basic comparators], page 116, for the details of comparators.

Updaters

Note that the basic premise of mappings srfi is to treat mappings as immutable. Each updating operation comes with a purely functional version (without bang) and a linear update version (with bang), but the linear update version may not require to destructively modify the passed mapping; it’s merely a hint that it may reuse the argument for the efficiency. You always need to use the returned mapping as the result of update. If you use linear update versions, you shouldn’t use the passed mapping afterwards, for there’s no guarantee how the state of the passed mapping is.

mapping-adjoin m arg . . .
[Function]  
[R7RS mapping]  
{scheme.mapping}  
The arg . . . are alternating between key and value. Returns a mapping that contains all the entries in m plus given keys and values, with the same comparator as m. Linear update version mapping-adjoin! may destructively modify m to create the return value, while mapping-adjoin creates a new mapping.

Arguments are processed in order. If there’s already an entry in m with the same key as given to arg, the original entry remains.

(mapping-adjoin (mapping default-comparator 'a 1 'b 2) 'c 3 'a 4 'c 5)
⇒ mapping with a → 1, b → 2, c → 3

mapping-set  m arg . . .

 mapping-set!  m arg . . .

 [Function]  [Function]
 [R7RS mapping]  {scheme.mapping} The arg . . . are alternating between key and value. Returns a mapping that contains all the entries in m plus given keys and values, with the same comparator as m. Linear update version mapping-set! may destructively modify m to create the return value, while mapping-set creates a new mapping.

Arguments are processed in order. If there’s already an entry in m with the same key as given to arg, the new key-value pair supersedes the old one.

(mapping-set (mapping default-comparator 'a 1 'b 2) 'c 3 'a 4 'c 5)
⇒ mapping with a → 4, b → 2, c → 5

mapping-replace  m key value

 mapping-replace!  m key value

 [Function]  [Function]
 [R7RS mapping]  {scheme.mapping} If the mapping m has an entry of key, return a mapping with the value of the entry replaced for value. If m doesn’t have an entry with key, m is returned unchanged.

Linear update version mapping-replace! may destructively modify m to produce the return value, while mapping-replace creates a new mapping.

(mapping-replace (mapping default-comparator 'a 1 'b 2) 'a 3)
⇒ mapping with a → 3, b → 2

(mapping-replace (mapping default-comparator 'a 1 'b 2) 'c 3)
⇒ mapping with a → 1, b → 2

mapping-delete  m key . . .

 mapping-delete!  m key . . .

 [Function]  [Function]
 [R7RS mapping]  {scheme.mapping} Returns a mapping that is the same as m except its entries with any of the given key . . . being removed. Keys that are not in m are ignored.

Linear update version mapping-delete! may destructively modify m to produce the return value, while mapping-delete creates a new mapping.

mapping-delete-all  m key-list

 mapping-delete-all!  m key-list

 [Function]  [Function]
 [R7RS mapping]  {scheme.mapping} Returns a mapping that is the same as m except its entries with any of the given keys in key-list being removed. Keys that are not in m are ignored.

Linear update version mapping-delete-all! may destructively modify m to produce the return value, while mapping-delete-all creates a new mapping.

mapping-intern  m key make-value

 mapping-intern!  m key make-value

 [Function]  [Function]
 [R7RS mapping]  {scheme.mapping} Looks up key in the mapping m, and returns two values, m and the associated value. If m does not contain an entry with key, a thunk make-value is invoked, and creates a new mapping that contains all entries in m plus a new entry with key and the return value of make-value, then returns the new mapping and the return value of make-value.

Linear update version mapping-intern! may destructively modify m to produce the return value, while mapping-intern creates a new mapping.

(mapping-intern (mapping default-comparator 'a 1) 'b (\[] 2))
⇒
mapping with \( a \to 1, b \to 2 \)
and
2

\((\text{mapping-intern} \ (\text{mapping default-comparator} \ 'a \ 1) \ 'a \ (^\{\} \ 2))\)
\(\Rightarrow\)
mapping with \( a \to 1 \)
and
1

**mapping-update** \( m \ \text{key} \ \text{updater} \ :\text{optional} \ \text{failure} \ \text{success} \)

**mapping-update!** \( m \ \text{key} \ \text{updater} \ :\text{optional} \ \text{failure} \ \text{success} \)

[R7RS mapping] \{scheme.mapping\} Semantically equivalent to this:

\((\text{mapping-set} \ m \ \text{var} \ \ (\text{updater} \ (\text{mapping-ref} \ m \ \text{key} \ \text{failure} \ \text{success})))\)

The `failure` and `success` optional arguments are procedures with zero and one arguments, respectively. When omitted, `failure` defaults to a thunk that raises an error, and `success` defaults to `identity`.

First, \( \text{key} \) is looked up in \( m \). If an entry is found, the associated value is passed to `success`; otherwise, `failure` is called with no arguments. Either way, let the returned value be \( v0 \).

Then, \( v0 \) is passed to `updater`. Let its result be \( v1 \).

Finally, a mapping with the same entries as \( m \) except the value of \( \text{key} \) is altered to \( v1 \) is returned.

Linear update version `mapping-update!` may destructively modify \( m \) to produce the return value, while `mapping-update` creates a new mapping.

\((\text{mapping-update} \ (\text{mapping default-comparator} \ 'a \ 1) \ 'a \ (\text{pa}$ + 1))\)
\(\Rightarrow\) mapping with \( a \to 2 \)

\((\text{mapping-update} \ (\text{mapping default-comparator}) \ 'a \ (\text{pa}$ + 1) \ (^\{\} \ 0))\)
\(\Rightarrow\) mapping with \( a \to 1 \)

**mapping-update/default** \( m \ \text{key} \ \text{updater} \ \text{default} \)

**mapping-update!/default** \( m \ \text{key} \ \text{updater} \ \text{default} \)

[R7RS mapping] \{scheme.mapping\}

**mapping-pop** \( m \ :\text{optional} \ \text{failure} \)

**mapping-pop!** \( m \ :\text{optional} \ \text{failure} \)

[R7RS mapping] \{scheme.mapping\}

**mapping-search** \( m \ \text{k} \ \text{failure} \ \text{success} \)

**mapping-search!** \( m \ \text{k} \ \text{failure} \ \text{success} \)

[R7RS mapping] \{scheme.mapping\}

The whole mapping

**mapping-size** \( m \)

[R7RS mapping] \{scheme.mapping\}

**mapping-find** \( \text{pred} \ m \ \text{failure} \)

[R7RS mapping] \{scheme.mapping\}
mapping-count pred m  
[R7RS mapping] {scheme.mapping}

mapping-any? pred m 
[Function]

mapping-every? pred m  
[R7RS mapping] {scheme.mapping}

mapping-keys m  
[Function]

mapping-values m  
[R7RS mapping] {scheme.mapping}

mapping-entries m  
[R7RS mapping] {scheme.mapping}

Mapping and folding

mapping-map proc comparator m  
[R7RS mapping] {scheme.mapping}

mapping-map/monotone proc comparator m  
[R7RS mapping] {scheme.mapping}

mapping-map/monotone! proc comparator m  
[R7RS mapping] {scheme.mapping}

mapping-for-each proc m  
[R7RS mapping] {scheme.mapping}

mapping-fold kons knil m 
[R7RS mapping] {scheme.mapping}

mapping-fold/reverse kons knil m  
[R7RS mapping] {scheme.mapping}

mapping-map->list proc m  
[R7RS mapping] {scheme.mapping}

mapping-filter pred m 
[Function]

mapping-filter! pred m  
[R7RS mapping] {scheme.mapping}

mapping-remove pred m  
[Function]

mapping-remove! pred m  
[R7RS mapping] {scheme.mapping}

mapping-partition pred m 
[Function]

mapping-partition! pred m  
[R7RS mapping] {scheme.mapping}

Copying and conversion

mapping-copy m  
[R7RS mapping] {scheme.mapping}

mapping->alist m  
[R7RS mapping] {scheme.mapping}

alist->mapping comparator alist  
[R7RS mapping] {scheme.mapping}
alist->mapping! m alist
   [R7RS mapping] {scheme.mapping}

alist->mapping/ordered comparator alist
alist->mapping/ordered! m alist
   [R7RS mapping] {scheme.mapping}

Submappings

mapping=? comparator m1 m2 ...
   [R7RS mapping] {scheme.mapping}
mapping<? comparator m1 m2 ...
mapping<=? comparator m1 m2 ...
mapping>? comparator m1 m2 ...
mapping>=? comparator m1 m2 ...
   [R7RS mapping] {scheme.mapping}

Set operations

mapping-union m1 m2 ...
mapping-union! m1 m2 ...
   [R7RS mapping] {scheme.mapping}
mapping-intersection m1 m2 ...
mapping-intersection! m1 m2 ...
   [R7RS mapping] {scheme.mapping}
mapping-difference m1 m2 ...
mapping-difference! m1 m2 ...
   [R7RS mapping] {scheme.mapping}
mapping-xor m1 m2 ...
mapping-xor! m1 m2 ...
   [R7RS mapping] {scheme.mapping}

Mappings with ordered keys

mapping-min-key m
mapping-max-key m
   [R7RS mapping] {scheme.mapping}
mapping-min-value m
mapping-max-value m
   [R7RS mapping] {scheme.mapping}
mapping-min-entry m
mapping-max-entry m
   [R7RS mapping] {scheme.mapping}
mapping-key-predecessor m obj failure
mapping-key-successor m obj failure
   [R7RS mapping] {scheme.mapping}
### Comparators

- **make-mapping-comparator**: `comparator`  
  ```scheme
  [R7RS mapping] {scheme.mapping}
  ```

- **mapping-comparator**:  
  ```scheme
  [R7RS mapping] {scheme.mapping}
  ```

### 10.3.20.2 Hashmaps

#### Constructor

- **hashmap**: `comparator key value . . .`  
  ```scheme
  [R7RS mapping] {scheme.mapping.hash} Creates a new hashmap with the given `comparator`, whose initial content is provided by `key value . . .`.  
  The `comparator` argument must be a comparator (see Section 6.2.4 [Basic comparators], page 116).  
  The `key value . . .` arguments must be even length, alternating keys and values.
  ```
  ```scheme
  (define m (hasheam default-comparator 'a 1 'b 2))
  ```
  ```scheme
  (hasheam-ref m 'a) ⇒ 1
  (hasheam-ref m 'b) ⇒ 2
  ```

- **hashmap-unfold**: `p f g seed comparator`  
  ```scheme
  [R7RS mapping] {scheme.mapping.hash} Creates a new hashmap, whose content is populated by three procedures, `p`, `f` and `g`, and a seed value `seed`, as follows.
  In each iteration, we have a current seed value, whose initial value is `seed`.
  First, `p`, a stop predicate, is applied to the current seed value. If it returns true, we stop iteration and returns the new hashmap.
  Next, `f` is applied to the current seed value. It must return two values. The first one is for a key and the second one for the value. We add this pair to the hashmap.
  ```
Then, \( g \) is applied to the current seed value. The result becomes the seed value of the next iteration. And we iterate.

The following example creates a hashmap that maps ASCII characters to their character codes:

\[
\text{(hashmap-unfold (cut >= <> 128)
 \quad \; (^c \to (\text{values (integer->char c) c}))
 \quad \; (cut + <> 1)
 \quad \; 0
 \quad \; \text{default-comparator})}
\]

**Predicates**

- **hashmap?** \( \text{obj} \)  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} Returns \#t iff \( \text{obj} \) is a hashmap object.

- **hashmap-empty?** \( \text{m} \)  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} \( \text{M} \) must be a hashmap. Returns \#t if \( \text{m} \) is empty, \#f otherwise. In Gauche, this is same as \text{tree-map-empty}? (see Section 6.14.2 [Treemaps], page 212).

- **hashmap-contains?** \( \text{m} \) \( \text{key} \)  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} \( \text{M} \) must be a hashmap. Returns \#t if \( \text{m} \) has an entry with \( \text{key} \), \#f otherwise. In Gauche, this is same as \text{tree-map-exists}? (see Section 6.14.2 [Treemaps], page 212).

- **hashmap-disjoint?** \( \text{m1} \) \( \text{m2} \)  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} Returns \#t if two hashmaps \( \text{m1} \) and \( \text{m2} \) have no keys in common. In other words, there’s no such key \( \text{K} \) that satisfy both \( \text{hashmap-contains? m1 K} \) and \( \text{hashmap-contains? m2 K} \).

**Accessors**

- **hashmap-ref** \( \text{m} \) \( \text{key} \) :optional failure success  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} Get the value from a hashmap \( \text{m} \) associated with \( \text{key} \), and calls success on the value, and returns its result. If \( \text{m} \) doesn’t have \( \text{key} \), failure is invoked with no arguments and its result is returned. Both success and failure is called in tail context.

  When failure is omitted and key is not found, an error is signaled. When success is omitted, identity is assumed.

- **hashmap-ref/default** \( \text{m} \) \( \text{key} \) \( \text{default} \)  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} Returns the value associated to \( \text{key} \) from a hashmap \( \text{m} \). If \( \text{m} \) doesn’t have \( \text{key} \), \( \text{default} \) is returned.

- **hashmap-key-comparator** \( \text{m} \)  
  [Function]  
  [R7RS mapping] \{scheme.mapping.hash\} Returns a comparator used to compare keys in a hashmap \( \text{m} \). See Section 6.2.4 [Basic comparators], page 116, for the details of comparators.

**Updaters**

Note that the basic premise of hashmaps srfi is to treat hashmaps as immutable. Each updating operation comes with a purely functional version (without bang) and a linear update version (with bang), but the linear update version may not require to destructively modify the passed hashmap; it’s merely a hint that it may reuse the argument for the efficiency. You always need to use the returned hashmap as the result of update. If you use linear update versions, you
shouldn’t use the passed hashmap afterwards, for there’s no guarantee how the state of the passed hashmap is.

hashmap-adjoin \( m \) arg ...  
hashmap-adjoin! \( m \) arg ...  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-set \( m \) arg ...  
hashmap-set! \( m \) arg ...  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-replace \( m \) key value  
hashmap-replace! \( m \) key value  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-delete \( m \) key ...  
hashmap-delete! \( m \) key ...  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-delete-all \( m \) key-list  
hashmap-delete-all! \( m \) key-list  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-intern \( m \) key failure  
hashmap-intern! \( m \) key failure  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-update \( m \) key updater :optional failure success  
hashmap-update! \( m \) key updater :optional failure success  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-update/default \( m \) key updater default  
hashmap-update!/default \( m \) key updater default  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-pop \( m \) :optional failure  
hashmap-pop! \( m \) :optional failure  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-search \( m \) \( k \) failure success  
hashmap-search! \( m \) \( k \) failure success  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

The whole hashmap

hashmap-size \( m \)  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-find pred \( m \) failure  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-count pred \( m \)  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}

hashmap-any? pred \( m \)  
hashmap-every? pred \( m \)  
    [Function]  
    [R7RS mapping] \{scheme.mapping.hash\}
hashmap-keys m
hashmap-values m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-entries m  
  [R7RS mapping] {scheme.mapping.hash}

Mapping and folding

hashmap-map proc comparator m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-for-each proc m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-fold kons knil m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-map->list proc m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-filter pred m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-filter! pred m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-remove pred m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-remove! pred m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-partition pred m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap-partition! pred m  
  [R7RS mapping] {scheme.mapping.hash}

Copying and conversion

hashmap-copy m  
  [R7RS mapping] {scheme.mapping.hash}
hashmap->alist m  
  [R7RS mapping] {scheme.mapping.hash}
alist->hashmap comparator alist  
  [R7RS mapping] {scheme.mapping.hash}
alist->hashmap! m alist  
  [R7RS mapping] {scheme.mapping.hash}

Subhashmaps

hashmap=? comparator m1 m2 . . .  
  [R7RS mapping] {scheme.mapping.hash}
hashmap<? comparator m1 m2 . . .  
hashmap<=? comparator m1 m2 . . .  
hashmap?> comparator m1 m2 . . .  
hashmap=>? comparator m1 m2 . . .  
  [R7RS mapping] {scheme.mapping.hash}
Set operations

hashmap-union  \( m1 \ m2 \ldots \)  \[Function\]
hashmap-union!  \( m1 \ m2 \ldots \)  \[Function\]
  \[R7RS mapping\] \{scheme.mapping.hash\}

hashmap-intersection  \( m1 \ m2 \ldots \)  \[Function\]
hashmap-intersection!  \( m1 \ m2 \ldots \)  \[Function\]
  \[R7RS mapping\] \{scheme.mapping.hash\}

hashmap-difference  \( m1 \ m2 \ldots \)  \[Function\]
hashmap-difference!  \( m1 \ m2 \ldots \)  \[Function\]
  \[R7RS mapping\] \{scheme.mapping.hash\}

hashmap-xor  \( m1 \ m2 \ldots \)  \[Function\]
hashmap-xor!  \( m1 \ m2 \ldots \)  \[Function\]
  \[R7RS mapping\] \{scheme.mapping.hash\}

Comparators

make-hashmap-comparator  \texttt{comparator}  \[Function\]
  \[R7RS mapping\] \{scheme.mapping.hash\}

hashmap-comparator  \[Variable\]
  \[R7RS mapping\] \{scheme.mapping.hash\}

10.3.21 \texttt{scheme.division} - R7RS integer division

\texttt{scheme.division}  \[Module\]

This module provides a comprehensive set of integer division operators.

Quotient and remainder in integer divisions can be defined in multiple ways, when you consider the choice of sign of the result with regard to the operands. Gauche has built-in procedures in several flavors: R5RS quotient, remainder and modulo, R6RS div, mod, div0 and mod0, and R7RS floor-quotient, floor-remainder, floor/, truncate-quotient, truncate-remainder, truncate/.

This module complements R7RS procedures, by adding ceiling, round, euclidean and balanced variants.

The following procedures are in \texttt{scheme.division} but built-in in Gauche (see Section 6.3.4 [Arithmetics], page 126).

floor-quotient  floor-remainder  floor/
truncate-quotient  truncate-remainder  truncate/

ceiling-quotient  \( n \ d \)  \[Function\]
ceiling-remainder  \( n \ d \)  \[Function\]
ceiling/  \( n \ d \)  \[Function\]
  \[R7RS division\] \{scheme.division\}
  ceiling-quotient = ceiling(n / d)
  ceiling-remainder = n - d * ceiling-quotient
  ceiling/ = values(ceiling-quotient, ceiling-remainder)

round-quotient  \( n \ d \)  \[Function\]
round-remainder  \( n \ d \)  \[Function\]
round/ \( n \; d \) \[\text{[Function]}\]

[R7RS division] \{scheme.division\}

round-quotient = \( \text{round}(n/d) \)
round-remainder = \( n - d \times \text{round-quotient} \)
round/ = values(round-quotient, round-remainder)

euclidean-quotient \( n \; d \) \[\text{[Function]}\]
euclidean-remainder \( n \; d \) \[\text{[Function]}\]
euclidean/ \( n \; d \) \[\text{[Function]}\]

[R7RS division] \{scheme.division\}

euclidean-quotient = \( \text{floor}(n / d) \) if \( d > 0 \)
    \( \text{ceiling}(n / d) \) if \( d < 0 \)
euclidean-remainder = \( n - d \times \text{euclidean-quotient} \)
euclidean/ = values(euclidean-quotient, euclidean-remainder)

The Euclidean variant satisfies a property \( 0 \leq \text{remainder} < \text{abs}(d) \). These are the same as R6RS’s div, mod, and div-and-mod, except that they accept non-integers (see Section 6.3.4 [Arithmetics], page 126)

balanced-quotient \( n \; d \) \[\text{[Function]}\]
balanced-remainder \( n \; d \) \[\text{[Function]}\]
balanced/ \( n \; d \) \[\text{[Function]}\]

[R7RS division] \{scheme.division\}

balanced-quotient = \( \text{roundup}(n / d) \)
balanced-remainder = \( n - d \times \text{balanced-quotient} \)
balanced/ = values(balanced-quotient, balanced-remainder)

where \( \text{roundup}(x) \) rounds towards zero if \( |x| - \text{floor}(|x|) < 0.5 \),
    and away from zero if \( |x| - \text{floor}(|x|) \geq 0.5 \),

The balanced variant satisfies a property \( -\text{abs}(d/2) \leq \text{remainder} < \text{abs}(d/2) \). These are the same as R6RS’s div0, mod0, and div0-and-mod0, except that they accept non-integers (see Section 6.3.4 [Arithmetics], page 126).

10.3.22 scheme.bitwise - R7RS bitwise operations

scheme.bitwise \[\text{[Module]}\]

This module provides comprehensive bitwise operations. Originally it was SRFI-151. It is mostly a superset of SRFI-60, with some change of names for the consistency and the compatibility (see Section 11.14 [Integers as bits], page 718). We keep SRFI-60 for legacy code, while recommend this module to be used in the new code.

The following procedures are Gauche built-in. See Section 6.3.6 [Basic bitwise operations], page 136, for the description.

\begin{verbatim}
integer-length  copy-bit  bit-field
\end{verbatim}

Basic operations

bitwise-not \( n \) \[\text{[Function]}\]

[R7RS bitwise] \{scheme.bitwise\} Returns the bitwise complement of \( n \). Same as built-in lognot (see Section 6.3.6 [Basic bitwise operations], page 136).

bitwise-and \( n \ldots \) \[\text{[Function]}\]

bitwise-ior \( n \ldots \) \[\text{[Function]}\]

bitwise-xor \( n \ldots \) \[\text{[Function]}\]
Chapter 10: Library modules - R7RS standard libraries

bitwise-eqv n . . .

[Function] {scheme.bitwise} When no arguments are given, these procedures returns -1, 0, 0 and -1, respectively. With one arguments, they return the argument as is. With two arguments, they return bitwise and, ior, xor, and eqv (complement of xor). With three or more arguments, they apply binary operations associatively, that is,

\[(\text{bitwise-xor } a \ b \ c) \equiv (\text{bitwise-xor } a \ (\text{bitwise-xor } b \ c))\]
\[\equiv (\text{bitwise-xor } (\text{bitwise-xor } a \ b) \ c)\]

Be careful that multi-argument bitwise-eqv does not produce bit 1 everywhere that all the argument’s bit agree.

The first three procedures are the same as built-in logand, logior and logxor, respectively (see Section 6.3.6 [Basic bitwise operations], page 136).

bitwise-nand n0 n1
bitwise-nor n0 n1
bitwise-andc1 n0 n1
bitwise-andc2 n0 n1
bitwise-orc1 n0 n1
bitwise-orc2 n0 n1

[Function] {scheme.bitwise} These operations are not associative.

\[\text{nand } n0 \ n1 \equiv (\text{NOT (AND } n0 \ n1))\]
\[\text{nor } n0 \ n1 \equiv (\text{NOT (OR } n0 \ n1))\]
\[\text{andc1 } n0 \ n1 \equiv (\text{AND (NOT } n0) \ n1)\]
\[\text{andc2 } n0 \ n1 \equiv (\text{AND } n0 \ (\text{NOT } n1))\]
\[\text{orc1 } n0 \ n1 \equiv (\text{OR (NOT } n0) \ n1)\]
\[\text{orc2 } n0 \ n1 \equiv (\text{OR } n0 \ (\text{NOT } n1))\]

Integer operations

arithmetic-shift n count

[Function] {scheme.bitwise} Shift n for count bits to left; if count is negative, it shifts n to right for -count bits.

Same as built-in ash (see Section 6.3.6 [Basic bitwise operations], page 136).

bit-count n

[Function] {scheme.bitwise} If n is positive, returns the number of 1’s in n. If n is negative, returns the number of 0’s in n.

Same as built-in logcount (see Section 6.3.6 [Basic bitwise operations], page 136).

bitwise-if mask n0 n1

[Function] {scheme.bitwise} Returns integer, whose n-th bit is taken as follows: If the n-th bit of mask is 1, the n-th bit of n0; otherwise, the n-th bit of n1.

\[(\text{bitwise-if } \#b10101100 \ #b00110101 \ #b11001010)\]
\[\Rightarrow \#b01100110\]

Single-bit operations

bit-set? index n

[Function] {scheme.bitwise} Returns #t or #f if index-th bit (counted from LSB) of n is 1 or 0, respectively.

Same as built-in logbit? (see Section 6.3.6 [Basic bitwise operations], page 136).
bit-swap \(index_1\) \(index_2\) \(n\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns an integer with \(index_1\)-th bit and \(index_2\)-th bit are swapped. Index is counted from LSB.

any-bit-set? \(mask\) \(n\)  
[Function]  
every-bit-set? \(mask\) \(n\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns \(\#t\) iff any/all bits set in \(mask\) are also set in \(n\).  
any-bit-set? is the same as built-in logtest, except logtest accepts one or more arguments (see Section 6.3.6 [Basic bitwise operations], page 136).

first-set-bit \(n\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns the number of factors of two of integer \(n\); that is, returns a maximum \(k\) such that \((\text{expt}\ 2 \ k)\) divides \(n\) without a remainder. It is the same as the index of the least significant \(1\) in \(n\), hence the alias first-set-bit.

\[
\begin{align*}
(\text{first-set-bit} \ 0) & \Rightarrow -1 \quad ; \text{edge case} \\
(\text{first-set-bit} \ 1) & \Rightarrow 0 \\
(\text{first-set-bit} \ 2) & \Rightarrow 1 \\
(\text{first-set-bit} \ 15) & \Rightarrow 0 \\
(\text{first-set-bit} \ 16) & \Rightarrow 4
\end{align*}
\]

This is equivalent to Gauche’s built-in twos-exponent-factor (see Section 6.3.6 [Basic bitwise operations], page 136).

Bit field operations

bit-field-any? \(n\) \(start\) \(end\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns \(\#t\) iff any/all bits of \(n\) from \(start\) (inclusive) to \(end\) (exclusive) are set.

bit-field-every? \(n\) \(start\) \(end\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns \(n\) with the bits from \(start\) (inclusive) to \(end\) (exclusive) are set to all 0's/1's.

bit-field-replace \(dst\) \(src\) \(start\) \(end\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns \(dst\) with the bitfield from \(start\) to \(end\) are replaced with the least-significant \((\text{end} - \text{start})\) bits of \(src\).

\[
(\text{bit-field-replace} \ #b101010 \ #b010 \ 1 \ 4) \Rightarrow \ #b100100
\]

Same as built-in copy-bit-field (see Section 6.3.6 [Basic bitwise operations], page 136).

bit-field-replace-same \(dst\) \(src\) \(start\) \(end\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Returns \(dst\) with the bitfield from \(start\) to \(end\) are replaced with the src's bitfield from \(start\) to \(end\).

\[
(\text{bit-field-replace-same} \ #b111111 \ #b100100 \ 1 \ 4) \Rightarrow \ #b110101
\]

bit-field-rotate \(n\) \(count\) \(start\) \(end\)  
[Function]  
[R7RS bitwise] \{scheme.bitwise\} Rotate the region of \(n\) between \(start\)-th bit (inclusive) and \(end\)-th bit (exclusive) by \(count\) bits to the left. If \(count\) is negative, it rotates to the right by \(-\text{count}\) bits.

\[
\text{(bit-field-rotate} \ #b110100100010000 \ -1 \ 5 \ 9) \Rightarrow \ 26768 \ ;\#b1101000100010000
\]

\[
(\text{bit-field-rotate} \ #b1101001000010000 \ 1 \ 5 \ 9) \Rightarrow \ 26672 \ ;\#b110100000110000
\]
bit-field-reverse n start end  
[R7RS bitwise] {scheme.bitwise} Reverse the order of bits of n between start-th bit (inclusive) and end-th bit (exclusive).

(bit-field-reverse #b10100111 0 8)  
⇒ 229 ; #b1100101

Bits conversion

bits->list n :optional len  
[Function]

bits->vector n :optional len  
[Function]

[R7RS bitwise] {scheme.bitwise} Returns a list/vector of booleans of length len, corresponding to each bit in non-negative integer n, LSB-first. When len is omitted, (integer-length n) is used.

(bits->vector #b101101110)  
⇒ #(#f #t #t #t #f #t #t #f #t)

Note: SRFI-60 has a similar integer->list, but the order of bits is reversed.

list->bits bool-list  
[Function]

vector->bits bool-vector  
[Function]

[R7RS bitwise] {scheme.bitwise} Returns an exact integer formed from boolean values in given list/vector, LSB first. The result will never be negative.

(list->bits '(#f #t #t #t #f #t #t #f #t))  
⇒ #b101101110

Note: SRFI-60 has a similar list->integer, but the order of bits is reversed.

bits bool ...  
[Function]

[R7RS bitwise] {scheme.bitwise} Returns the integer coded by bools, LSB first. The result will never be negative.

(bits #f #t #t #t #f #t #t #f #t)  
⇒ #b101101110

Note: SRFI-60 has a similar booleans->integer, but the order of bits is reversed.

Fold, unfold and generate

bitwise-fold kons knil n  
[Function]

[R7RS bitwise] {scheme.bitwise} Traverse bits in integer n from LSB to the (integer-length n) bit, applying kons on the bit as boolean and the seed value, whose initial value is given by knil. Returns the last result of kons.

(bitwise-fold cons () #b10110111)  
⇒ (#t #f #t #t #f #t #t #t)

bitwise-for-each proc n  
[Function]

[R7RS bitwise] {scheme.bitwise} Applies proc to the bit as boolean in n, from LSB to the (integer-length n) bit. The result is discarded.

bitwise-unfold p f g seed  
[Function]

[R7RS bitwise] {scheme.bitwise} Generates a non-negative integer bit by bit, from LSB to MSB. The seed gives the initial state value. For each iteration, p is applied to the current state value, and if it returns a true value, the iteration ends and bitwise-unfold returns the accumulated bits as an integer. Otherwise, f is applied to the current state value, and its result, coerced to a boolean value, determines the bit value. Then g is applied to the current state value to produce the next state value of the next iteration.
The following expression produces a bitfield of width 100, where n-th bit indicates whether n is prime or not:

```
(use math.prime)
(bitwise-unfold (cut = 100 <>)
  small-prime?
  (cut + 1 <>)
  0)
```

**make-bitwise-generator n**

[Function] [R7RS bitwise] {scheme.bitwise} Returns a generator that generates boolean values corresponding to the bits in n, LSB-first. The returned generator is infinite.

This is similar to `bits->generator` in `gauche.generator`, except that the generator created by it stops at the integer length of n (see Section 9.12 [Generators], page 436).

### 10.3.23 scheme.fixnum - R7RS fixnums

**scheme.fixnum**

[Module] This module provides a set of fixnum-specific operations. Originally defined as SRFI-143.

A fixnum is a small exact integer that can be handled very efficiently. In Gauche, fixnum is 62bit wide on 64bit platforms, and 30bit wide on 32bit platforms.

Note that these procedures are defined only to work on fixnums, but it is not enforced. If you pass non-fixnum arguments, or the result falls out of range of fixnums, what happens is up to the implementation. Consider these procedures as the way to tell your intentions to the compiler for potential optimizations.

In the current Gauche architecture, generic numeric operators are just as efficient, so most procedures provided in this module are aliases to corresponding operators. However, we might employ some optimizations in future versions.

The procedure `fixnum?` is built-in, and not explained here. See Section 6.3.2 [Numerical predicates], page 123.

**fx-width**

[Variable] [R7RS fixnum] {scheme.fixnum} A variable bound to an exact positive integer w, where w is the greatest number such that exact integers between $2^{(w-1)} - 1$ and $-2^{(w-1)}$ are all fixnums. This value is the same as the built-in procedure `fixnum-width` returns (see Section 6.3.4 [Arithmetics], page 126).

In Gauche, it is usually 30 for 32bit platforms, and 62 for 64bit platforms.

**fx-greatest**

[Variable] [R7RS fixnum] {scheme.fixnum} Variables bound to the greatest fixnum and the least fixnum. They are the same as the built-in procedures `greatest-fixnum` and `least-fixnum` return, respectively (see Section 6.3.4 [Arithmetics], page 126).

The following table shows the typical values on Gauche:

<table>
<thead>
<tr>
<th>Platform</th>
<th>fx-greatest</th>
<th>fx-least</th>
</tr>
</thead>
<tbody>
<tr>
<td>32bit</td>
<td>536,870,911</td>
<td>-536,870,912</td>
</tr>
<tr>
<td>64bit</td>
<td>2,305,843,009,213,693,951</td>
<td>-2,305,843,009,213,693,952</td>
</tr>
</tbody>
</table>

**fx=? i ...**

[Function]

**fx<? i ...**

[Function]

**fx<=? i ...**

[Function]

**fx>? i ...**

[Function]
fx>=? i ...  [Function]
[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in =, <, <=, and >=, except that you should use these only for fixnums.

fxzero? i  [Function]
fxpositive? i  [Function]
fxnegative? i  [Function]
fxodd? i  [Function]
fxeven? i  [Function]
[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in zero?, positive?, negative?, odd? and even?, except that you should use these only for fixnums.

fxmax i j ...  [Function]
fxmin i j ...  [Function]
[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in max and min, except that you should use these only for fixnums.

fx+ i j  [Function]
fx- i j  [Function]
fx* i j  [Function]
[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in +, - and *, except that these take exactly two arguments, and you should use these only for fixnums and when the result fits within fixnum range.

fxneg i  [Function]
[R7RS fixnum] {scheme.fixnum} This is equivalent to single-argument -, except that you should use this only for fixnums and when the result fits within fixnum range.

fxquotient i j  [Function]
fxremainder i j  [Function]
fxabs i  [Function]
fxsquare i  [Function]
[R7RS fixnum] {scheme.fixnum} These are equivalent to built-in quotient, remainder, abs and square, except that you should use these only for fixnums and when the result fits within fixnum range.

fxsqrt i  [Function]
[R7RS fixnum] {scheme.fixnum} This is equivalent to exact-integer-sqrt (not sqrt), except that you should use it only for fixnums. See Section 6.3.4 [Arithmetics], page 126.

fx+/carry i j k  [Function]
fx-/carry i j k  [Function]
fx*/carry i j k  [Function]
[R7RS fixnum] {scheme.fixnum} These calculates (+ i j k), (- i j k) and (+ (* i j) k), respectively, then split the result to the remainder value R in the fixnum range, and spilled value Q, and return those values. That is, (+ (* Q (expt 2 fx-width)) R) is the result of above calculations. Both Q and R fits in the fixnum range, and -2^(-(w-1)) <= R < 2^(-(w-1)), where w is fx-width.

(fx*/carry 1845917459 19475917581 4735374) ⇒ -942551854601421179 and 8

(+ (* 8 (expt 2 fx-width)) -942551854601421179) ⇒ 35950936292817682053
(+ (* 1845917459 19475917581) 4735374)  
⇒ 35950936292817682053

These are primitives to implement extended-precision integer arithmetic on top of fixnum operations. In Gauche, however, you can just use built-in bignums. We provide these for the compatibility.

- `fxnot i`  
- `fxand i ...`  
- `fxior i ...`  
- `fxxor i ...`  
- `fxarithmetic-shift i count`  
- `fxlength i`  
- `fxbit-count i`  
- `fxcopy-bit index i boolean`  
- `fxbit-set? index i`  
- `fxbit-field i start end`  
- `fxfirst-set-bit i`

[R7RS fixnum] {scheme.fixnum} These are equivalent to `lognot`, `logand`, `logior`, `logxor`, `ash`, `integer-length`, `logcount`, `copy-bit`, `logbit?`, `bit-field`, and `twos-exponent-factor` respectively, except that you should use these only for fixnums. See Section 6.3.6 [Basic bitwise operations], page 136.

- `fxif mask i j`  
- `fxbit-field-rotate i start end`  
- `fxbit-field-rotate i start end`

[R7RS fixnum] {scheme.fixnum} These are equivalent to SRFI-60's `bitwise-if`, `rotate-bit-field` and `reverse-bit-field`, except that you should use these only for fixnums. See Section 11.14 [Integers as bits], page 718.

### 10.3.24 scheme.flonum - R7RS flonum

scheme.flonum  
[Module]

This module provides a set of flonum-specific operations. Originally defined as SRFI-144.

In Gauche, a flonum is IEEE 754 double-precision floating point numbers.

Note that these procedures are defined only to work on flonums, but it is not enforced. If you pass non-flonum arguments, the result is undefined. Consider these procedures as the way to tell your intentions to the compiler for potential optimizations.

In the current Gauche architecture, generic numeric operators are just as efficient, so most procedures provided in this module are aliases to corresponding operators. However, we might employ some optimizations in future versions.

The procedure `flonum?` is built-in, and not explained here. See Section 6.3.2 [Numerical predicates], page 123.

**Constants**

We also have a few constants in `math.const` module (see Section 12.35 [Mathematical constants], page 893).

- `fl-e`  

[R7RS flonum] {scheme.flonum} The base of natural logarithm e.

Gauche also has a constant `e` in `math.const` (see Section 12.35 [Mathematical constants], page 893).
fl-1/e
[Constant] [R7RS flonum] {scheme.flonum} (/ e).

fl-e-2
[Constant] [R7RS flonum] {scheme.flonum} (square e).

fl-e-pi/4
[Constant] [R7RS flonum] {scheme.flonum} (expt e (/ pi 4)).

fl-log2-e
[Constant] [R7RS flonum] {scheme.flonum} (log2 e). (Same as (/ (log 2))).

fl-log10-e
[Constant] [R7RS flonum] {scheme.flonum} (log10 e). (Same as (/ (log 10))).

fl-log-2
[Constant] [R7RS flonum] {scheme.flonum} (log 2).

fl-1/log-2
[Constant] [R7RS flonum] {scheme.flonum} (/ (log 2)). (Same as (log2 e)).

fl-log-3
[Constant] [R7RS flonum] {scheme.flonum} (log 3).

fl-log-pi
[Constant] [R7RS flonum] {scheme.flonum} (log pi).

fl-log-10
[Constant] [R7RS flonum] {scheme.flonum} (log 10).

fl-1/log-10
[Constant] [R7RS flonum] {scheme.flonum} (/ (log 10)). (Same as (log10 e)).

fl-pi
[Constant] [R7RS flonum] {scheme.flonum} pi.

fl-1/pi
[Constant] [R7RS flonum] {scheme.flonum} (/ pi).

fl-2pi
[Constant] [R7RS flonum] {scheme.flonum} (* 2 pi).

fl-pi/2
[Constant] [R7RS flonum] {scheme.flonum} (/ pi 2).

fl-pi/4
[Constant] [R7RS flonum] {scheme.flonum} (/ pi 4).

fl-pi-squared
[Constant] [R7RS flonum] {scheme.flonum} (square pi).

fl-degree
[Constant] [R7RS flonum] {scheme.flonum} (/ pi 180).

fl-2/pi
[Constant] [R7RS flonum] {scheme.flonum} (/ 2 pi).

fl-2/sqrt-pi
[Constant] [R7RS flonum] {scheme.flonum} (/ 2 (sqrt pi)).
Chapter 10: Library modules - R7RS standard libraries

\[ \text{fl-sqrt-2} \]
\[ (\text{sqrt} 2). \]

\[ \text{fl-sqrt-3} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{sqrt} 3). \]

\[ \text{fl-sqrt-5} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{sqrt} 5). \]

\[ \text{fl-sqrt-10} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{sqrt} 10). \]

\[ \text{fl-1/sqrt-2} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (/ (\text{sqrt} 2)). \]

\[ \text{fl-cbrt-2} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{expt} 2 1/3). \]

\[ \text{fl-cbrt-3} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{expt} 3 1/3). \]

\[ \text{fl-4thrt-2} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{expt} 2 1/4). \]

\[ \text{fl-phi} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (/ (+ 1 (\text{sqrt} 5)) 2). \]

\[ \text{fl-log-phi} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{log} \text{fl-phi}). \]

\[ \text{fl-1/log-phi} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (/ (\text{log} \text{fl-phi})). \]

\[ \text{fl-euler} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} \text{Euler's constant}. \]

\[ \text{fl-e-euler} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{exp} \text{fl-euler}) \]

\[ \text{fl-sin-1} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{sin} 1) \]

\[ \text{fl-cos-1} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{cos} 1) \]

\[ \text{fl-gamma-1/2} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{gamma} 1/2) \]

\[ \text{fl-gamma-1/3} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{gamma} 1/3) \]

\[ \text{fl-gamma-2/3} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{gamma} 2/3) \]

\[ \text{fl-gamma-1/3} \]
\[ \text{R7RS flonum} \{ \text{scheme.flonum} \} (\text{gamma} 1/3) \]
constant fl-greatest

[R7RS flonum] {scheme.flonum} Bound to the largest/smallest positive finite flonum. The latter is the same value as returned from (flonum-min-denormalized) in Gauche (see Section 6.3.3 [Numerical comparison], page 125).

constant fl-least

[R7RS flonum] {scheme.flonum} The same as (flonum-epsilon) in Gauche (see Section 6.3.3 [Numerical comparison], page 125).

constant fl-epsilon

[R7RS flonum] {scheme.flonum} The same as (flonum-epsilon) in Gauche (see Section 6.3.3 [Numerical comparison], page 125).

constant fl-fast-fl+*

[R7RS flonum] {scheme.flonum} If this is #t, (fl+* x y z) executes as fast as, or faster than, (fl+ (fl* x y) z). If not, #f.

constant fl-integer-exponent-zero
constant fl-integer-exponent-nan

[R7RS flonum] {scheme.flonum} These are exact integer values returned in special occasion from flinteger-exponent. The values themselves don’t mean much; they’re to be compared with the result of flinteger-exponent.

If its argument is 0, flinteger-exponent returns fl-integer-exponent-zero. If its argument is *nan.0, flinteger-exponent returns fl-integer-exponent-nan.

Constructors

flonum number

[R7RS flonum] {scheme.flonum} Returns a flonum that’s equal to or (if there’s no equivalent flonum) the closest to number. If number is not a real number, +nan.0 is returned. (Note: SRFI-144 recommends it, but a conformant implementation may signal an error in such case. Portable code shouldn’t rely on this behavior.)

fladjacent x y

[R7RS flonum] {scheme.flonum} Returns a flonum adjacent to x in the direction of y. If x = y, x is returned.

(fladjacent 1.0 2.0) ⇒ 1.0000000000000002
(fladjacent 1.0 0.0) ⇒ 0.9999999999999999

flcopy-sign x y

[R7RS flonum] {scheme.flonum} Returns a flonum whose absolute value is (abs x) and whose sign is the sign of y.

make-flonum x n

[R7RS flonum] {scheme.flonum} Returns a flonum (* x (expt 2 n)). It is the same as ldexp (see Section 6.3.5 [Numerical conversions], page 134).

Accessors

fl-integer-fraction x

[R7RS flonum] {scheme.flonum} Returns two flonums, the integral part of x and the fractional part of x. Same as modf (see Section 6.3.5 [Numerical conversions], page 134).

If x is +inf.0, +inf.0 and 0.0 is returned. If x is -inf.0, -inf.0 and -0.0 is returned. If x is +nan.0, +nan.0 and +nan.0 is returned. (These corner cases are not explicit in SRFI-144, but follows the POSIX specification of modf.)

(flinteger-fraction fl-pi) ⇒ 3.0 and 0.14159265358979312
flexponent x  
[R7RS flonum] {scheme.flonum} Returns the exponent part of x as a flonum. If x is a non-zero finite value, the result is an integer.
If x is zero, -inf.0 is returned. If x is +inf.0 or -inf.0, +inf.0 is returned.

(flexponent 1.0)  ⇒  0.0
(flexponent 1024.0)  ⇒  10.0
(flexponent 0.01)  ⇒  -7.0
(flexponent fl-least)  ⇒  -1074.0

flinteger-exponent x  
[R7RS flonum] {scheme.flonum} Returns the same as flexponent but as an exact integer.
If x is zero, the value of fl-integer-exponent-zero is returned. If x is +inf.0 or -inf.0, a large implementation-dependent exact integer (usually it’s so large that (ldexp 1 (flinteger-exponent +inf.0)) becomes +inf.0) is returned. If x is +nan.0, the value of fl-integer-exponent-nan is returned.

flnormalized-fraction-exponent x  
[R7RS flonum] {scheme.flonum} Returns two values, a normalized mantissa of x with the same sign as x as a flonum, and an exact integer exponent of x. If it returns a value y and n, x = (* y (expt 2 n)) and x = (ldexp y n). This is the same as frexp (see Section 6.3.5 [Numerical conversions], page 134).
If x is non-zero finite value, the first value falls between 0.5 (inclusive) and 1.0 (exclusive). The corner cases are not explicit in SRFI-144, but Gauche follows frexp: If x is (minus) zero, it returns (minus) zero and 0; if x is infinity, it returns infinity (of the same sign) and 0; if x is +nan.0, it returns +nan.0 and 0.

(flnormalized-fraction-exponent 12345.6789)  ⇒  0.7535204406738282 and 14
(make-flonum 0.7535204406738282 14)  ⇒  12345.6789

flsign-bit x  
[R7RS flonum] {scheme.flonum} Returns 0 if x is positive or 0.0, and 1 if it is negative (including -0.0). (flsign-bit +nan.0) is implementation-dependent.

There’s also flsgn, if you need sign instead of a bit value.

Predicates
(Note: flonum? is built-in; see Section 6.3.2 [Numerical predicates], page 123).

fl=?  x y z . . .  
fl<?  x y z . . .  
fl>?  x y z . . .  
fl<=?  x y z . . .  
fl>=?  x y z . . .  

[R7RS flonum] {scheme.flonum} Flonum specific version of numerical comparison predicates =, <, <= and >, respectively.
Currently these are just an alias of the generic numerical comparison predicates; hence they don’t reject when you pass non-flonum arguments, but doing so is not portable. These are to give compilers hints that you are passing flonums so that it can optimize.
Note that -0.0 and 0.0 are the same in terms of these predicates, e.g. (fl<? -0.0 0.0) is #f.
If a +nan.0 is passed to any of the arguments, these predicates returns #f.
flunordered? x y

([R7RS flonum] {scheme.flonum}) Returns #t if at least one of x or y is a +nan.0.

flinteger? x

([R7RS flonum] {scheme.flonum}) Returns #t if x is an integral flonum, #f otherwise.

flzero? x
flpositive? x
flnegative? x

([R7RS flonum] {scheme.flonum}) Flonum-specific version of zero?, positive? and negative?. Note that (flnegative? -0.0) is #f. You need flsign-bit or flsgn to distinguish negative zero from zero.

flodd? x
fleven? x

([R7RS flonum] {scheme.flonum}) Flonum-specific version of odd? and even?. An error is thrown if x is not an integer.

flfinite? x
flinfinite? x
flnan? x

([R7RS flonum] {scheme.flonum}) Flonum-specific version of finite?, infinite? and nan? (see Section 6.3.2 [Numerical predicates], page 123).

flnormalized? x
fldenormalized? x

([R7RS flonum] {scheme.flonum}) Returns #t iff x is a normalized/denormalized flonum, respectively.

Arithmetic

flmax x . . .
flmin x . . .

([R7RS flonum] {scheme.flonum}) Flonum-specific version of min and max, except that these can take no arguments, in which case -inf.0 and +inf.0 are returned, respectively.

Note that we don’t check whether the arguments are flonums or not, but passing non-flonums are not portable.

fl+ x . . .
fl* x . . .

([R7RS flonum] {scheme.flonum}) Flonum-specific version of +. and *. (see Section 6.3.4 [Arithmetics], page 126).

Note that we don’t check whether the arguments are flonums or not, but passing non-flonums is not portable.

fl++ x y z

([R7RS flonum] {scheme.flonum}) Returns (+ (* x y) z), but (potentially) faster, and more accurately. It calls C99 fma function internally. “More accurately” means that calculation is done in a single step, with rounding once at the end.

Note: As of release of 0.9.8, MinGW implementation of fma seems to produce slightly off value occasionally, as if it is calculated with the two steps (rounding after multiplication).
Chapter 10: Library modules - R7RS standard libraries

```
fl- x y ...  [Function]
fl- x y ...  [Function]
[R7RS flonum] {scheme.flonum} Flonum-specific version of -. and /. (see Section 6.3.4 [Arithmetics], page 126).
Note that we don’t check whether the arguments are flonums or not, but passing non-flonums is not portable.

flabs x  [Function]
[R7RS flonum] {scheme.flonum} Flonum-specific version of abs (see Section 6.3.4 [Arithmetics], page 126).
Note that we don’t check whether the argument is a flonum or not, but passing non-flonum is not portable.

flabsdiff x y  [Function]
[R7RS flonum] {scheme.flonum} Returns (abs (- x y)).
Note that we don’t check whether the arguments are flonums or not, but passing non-flonum is not portable.

flposdiff  [Function]
[R7RS flonum] {scheme.flonum} Returns (max (- x y) 0).
Note that we don’t check whether the arguments are flonums or not, but passing non-flonum is not portable.

flsgn x  [Function]
[R7RS flonum] {scheme.flonum} Returns 1.0 if x’s sign bit is 0 (zero or positive), -1.0 if it’s 1 (negative). Same as (flcopiesign 1.0 x), or (if (zero? (flsign-bit x)) 1.0 -1.0).
Note that (flsgn 0.0) is 1.0, while (flsgn -0.0) is -1.0. The result of passing +nan.0 is implementation-dependent, reflecting the sign bit of underlying representation; it returns either 1.0 or -1.0 but you can’t count on which.
To extract the sign bit, instead of obtaining a signed flonum, you can use flsign-bit.

flnumerator x  [Function]
fldenominator x  [Function]
[R7RS flonum] {scheme.flonum} Flonum-specific version of numerator and denominator (see Section 6.3.4 [Arithmetics], page 126).
For infinity and zero, denominator is 1.0.
Note that we don’t check whether the arguments are flonums or not, but passing non-flonum is not portable.

flfloor x  [Function]
flceiling x  [Function]
flround x  [Function]
fltruncate x  [Function]
[R7RS flonum] {scheme.flonum} Flonum-specific version of floor, ceiling, round and truncate (see Section 6.3.4 [Arithmetics], page 126).
Note that we don’t check whether the arguments are flonums or not, but passing non-flonum is not portable.

Exponents and logarithms

flexp x  [Function]
[R7RS flonum] {scheme.flonum} Flonum-specific version of exp (see Section 6.3.4 [Arithmetics], page 126). Returns (expt fl-e x).
```
flexp2 \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns \((\text{expt} 2 x)\)

flexp-1 \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns \((- 1 (\text{expt} \text{fl-e} x))\), but is much more accurate when \(x\) is small. We call C’s \text{expm1} internally.

flsquare \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns \((\ast. x x)\)

flsqrt \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Flonum-specific version of \text{sqrt}.

flcbrt \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns cubic root of a flonum \(x\).

flhypot \(x y\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Calculates \((\text{sqrt} (\ast. x x) (\ast. y y))\), with avoiding overflow or underflow during the intermediate steps.

flexpt \(x y\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Flonum-specific version of \text{expt} (see Section 6.3.4 [Arithmetics], page 126).

fllog \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Flonum-specific version of \text{log} (see Section 6.3.4 [Arithmetics], page 126).

fllog1+ \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns \((\text{log} (\ast. x 1))\), but is more accurate than \text{log} when \(x\) is near zero.

fllog2 \(x\) \[Function\] 
fllog10 \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns base-2 and base-10 logarithm of a flonum \(x\), respectively.

make-fllog-base \(x\) \[Function\] 
[R7RS flonum] \{scheme.flonum\} Returns a procedure that calculates base-\(x\) logarithm. An error is signalled if \(x\) isn’t a real number greater than 1.0.

\((\text{define} \text{log5} (\text{make-fllog-base} 5.0))\)

\((\text{log5} 25.0) \Rightarrow 5.0\)

**Trigonometric functions**

flsin \(x\) \[Function\] 
flcos \(x\) \[Function\] 
fltan \(x\) \[Function\] 
flasin \(x\) \[Function\] 
flacos \(x\) \[Function\] 
flatan \(x\) \[Function\] 
flatan \(y x\) \[Function\] 
flsinh \(x\) \[Function\] 
flcosh \(x\) \[Function\]
Chapter 10: Library modules - R7RS standard libraries

fltanh x
flasinh x
flacosh x
flatanh x

[R7RS flonum] {scheme.flonum} Flonum-specific version of sin, cos, tan, asin, acos, sinh, cosh, tanh, asinh, acosh, and atanh, respectively (see Section 6.3.4 [Arithmetics], page 126).

Note that flatan can take one or two arguments; if two arguments are passed, it calculates (atan (/ y x)).

Integer division

flquotient x y

[R7RS flonum] {scheme.flonum} Returns (fltruncate (fl/ x y)).

The result is always integer flonum, but x and y doesn’t need to be integers.

(flquotient 14.0 4.0) ⇒ 3.0
(flquotient -14.0 4.0) ⇒ -3.0
(flquotient 14.0 2.5) ⇒ 5.0
(flquotient -14.2 2.8) ⇒ -5.0

flremainder x y

[R7RS flonum] {scheme.flonum} Returns two values:

• Remainder of x divided by y. (Same as (flremainder x y), but we calculate it in different routines so the result may differ in a few ulp.

• Integer quotient of x divided by y, modulo 2^n (n >= 3), as an exact integer. Its sign is the same as x/y.

This corresponds to C99’s remquo.

This function is useful to reduce the input for the periodic functions with symmetries.

Special functions

flgamma x

[R7RS flonum] {scheme.flonum} Computes the value of gamma function for a flonum x. When x is integer, it is the same as the factorial of x-1.

This is same as Gauche’s built-in gamma (see Section 6.3.4 [Arithmetics], page 126).

flloggamma x

[R7RS flonum] {scheme.flonum} Returns two values, (log (abs (flgamma x))) and the sign of (flgamma x) as 1.0 if it is positive and -1.0 if it is negative.

The first value is calculated by Gauche’s built-in lgamma (see Section 6.3.4 [Arithmetics], page 126). It’s more accurate than using gamma then calling log. The second value is +nan.0 when x is -inf.0 or +nan.0.

flfirst-bessel n x

[R7RS flonum] {scheme.flonum} Returns the n-th order Bessel function of the first kind.
flsecond-bessel n x  
[R7RS flonum] \{scheme.flonum\} Returns the n-th order Bessel function of the second kind.

flerf x  
[R7RS flonum] \{scheme.flonum\} Returns the error function erf(x).

flerfc x  
[R7RS flonum] \{scheme.flonum\} Returns the complementary error function, 1 - erf(x).

### 10.3.25 scheme.bytevector - R7RS bytevectors

This module is taken from R6RS ([rnrs bytevectors]) (note that R7RS uses singular form). The bytevector in this module is u8vector in Gauche.

The following procedures are the same as in `gauche.uvector`. See Section 9.38.5 [Bytevector compatibility], page 569, for the explanations.

- `bytevector?`  
- `make-bytevector`  
- `bytevector-length`
- `bytevector=?`  
- `bytevector-fill!`  
- `bytevector-copy`
- `bytevector-u8-ref`  
- `bytevector-u8-set!`  
- `bytevector-s8-ref`
- `bytevector-s8-set!`  
- `bytevector->u8-list`  
- `u8-list->bytevector`

The following procedures are the same as in `gauche.unicode`. See Section 9.37 [Unicode utilities], page 549, for the explanations.

- `string->utf8`  
- `string->utf16`  
- `string->utf32`
- `utf8->string`  
- `utf16->string`  
- `utf32->string`

This module exports `bytevector-copy!`, which takes arguments in different order from R7RS base’s (and `gauche.uvector`’s) `bytevector-copy!`. It is the same as `bytevector-copy!-r6` in `gauche.uvector` (see Section 9.38.5 [Bytevector compatibility], page 569).

### endianness symbol

[R7RS bytevector] If `symbol` is a valid endianness symbol, returns it. Otherwise, raise an error at macro-expansion time. Useful to catch an error early, and explicitly indicates you mean endianness.

Valid symbols are listed in Section 6.3.7 [Endianness], page 138. Only `big` and `little` are portable.

native-endianness  
[R7RS bytevector] Returns a symbol representing the native endianness. Same as Gauche’s built-in `native-endian` (see Section 6.3.7 [Endianness], page 138).

bytevector-uint-ref bv pos endian size  
[Function]

bytevector-sint-ref bv pos endian size  
[Function]

[R7RS bytevector] Read size octets from u8vector `bv`, starting from pos-th octet, as an unsigned or signed integer, respectively. Similar to `binary.io`’s `get-uint` and `get-sint`, with different argument order.

They accept all valid endianness symbols in Gauche, but the portable code should only use `big` and `little` for `endianness` argument.

bytevector-uint-set! bv pos val endian size  
[Function]

bytevector-sint-set! bv pos val endian size  
[Function]

[R7RS bytevector] Store an unsigned or signed integer `val` into an u8vector `bv` starting from pos-th octet, for size octets, respectively. Similar to `binary.io`’s `put-uint!` and `put-sint!`, with different argument order.

They accept all valid endianness symbols in Gauche, but the portable code should only use `big` and `little` for `endianness` argument.
Chapter 10: Library modules - R7RS standard libraries

[Function]
bytevector->uint-list bv endian size
[Function]
bytevector->sint-list bv endian size

[R7RS bytevector] Convert the u8vector \( bv \) to a list of unsigned or signed integer, each of which is represented by \( \text{size-octets} \). The length of \( bv \) must be a multiple of \( \text{size} \); otherwise an error is signaled.

They accept all valid endianness symbols in Gauche, but the portable code should only use \texttt{big} and \texttt{little} for \textit{endian} argument.

[Function]
uint-list->bytevector lis endian size
[Function]
sint-list->bytevector lis endian size

[R7RS bytevector] Lis must be a list of unsigned or signed integers, respectively. Creates a u8vector of length \((\ast \text{size (length lis)})\), and stores each integer into the u8vector using \text{size octets}, and returns the u8vector.

They accept all valid endianness symbols in Gauche, but the portable code should only use \texttt{big} and \texttt{little} for \textit{endian} argument.

[Function]
bytevector-u16-ref bv k endian
[Function]
bytevector-s16-ref bv k endian
[Function]
bytevector-u32-ref bv k endian
[Function]
bytevector-s32-ref bv k endian
[Function]
bytevector-u64-ref bv k endian
[Function]
bytevector-s64-ref bv k endian
[Function]
bytevector-ieee-single-ref bv k endian
[Function]
bytevector-ieee-double-ref bv k endian

[R7RS bytevector] Retrieve a numerical value from a u8vector \( bv \) starting at index \( k \), using \textit{endian}. These are the same as \texttt{get-u16}, \ldots in \texttt{binary.io}, except that all arguments are mandatory. (Note that \texttt{ieee-single} and \texttt{ieee-double} corresponds to \texttt{f32} and \texttt{f64}). See Section 12.1 [Binary I/O], page 808, for the details.

They accept all valid endianness symbols in Gauche, but the portable code should only use \texttt{big} and \texttt{little} for \textit{endian} argument.

[Function]
bytevector-u16-native-ref bv k
[Function]
bytevector-s16-native-ref bv k
[Function]
bytevector-u32-native-ref bv k
[Function]
bytevector-s32-native-ref bv k
[Function]
bytevector-u64-native-ref bv k
[Function]
bytevector-s64-native-ref bv k
[Function]
bytevector-ieee-single-native-ref bv k
[Function]
bytevector-ieee-double-native-ref bv k

[R7RS bytevector] Like bytevector-u16-ref etc., but uses native endianness.

[Function]
bytevector-u16-set! bv k val endian
[Function]
bytevector-s16-set! bv k val endian
[Function]
bytevector-u32-set! bv k val endian
[Function]
bytevector-s32-set! bv k val endian
[Function]
bytevector-u64-set! bv k val endian
[Function]
bytevector-s64-set! bv k val endian
[Function]
bytevector-ieee-single-set! bv k val endian
[Function]
bytevector-ieee-double-set! bv k val endian

[R7RS bytevector] Store a numerical value into a u8vector \( bv \) starting at index \( k \), using \textit{endian}. These are the same as \texttt{put-u16!}, \ldots in \texttt{binary.io}, except that all arguments are mandatory. (Note that \texttt{ieee-single} and \texttt{ieee-double} corresponds to \texttt{f32} and \texttt{f64}). See Section 12.1 [Binary I/O], page 808, for the details.
They accept all valid endianness symbols in Gauche, but the portable code should only use `big` and `little` for `endian` argument.

- `bytevector-u16-native-set! bv k val` [Function]
- `bytevector-s16-native-set! bv k val` [Function]
- `bytevector-u32-native-set! bv k val` [Function]
- `bytevector-s32-native-set! bv k val` [Function]
- `bytevector-u64-native-set! bv k val` [Function]
- `bytevector-s64-native-set! bv k val` [Function]
- `bytevector-ieee-single-native-set! bv k val` [Function]
- `bytevector-ieee-double-native-set! bv k val` [Function]

[R7RS bytevector] Like `bytevector-u16-set!` etc., but uses native endianness.

### 10.3.26 scheme.show - R7RS combinator formatting

#### Module structure

**scheme.show** [Module]
Exports bindings of R7RS `(scheme show)` library. From R7RS programs, those bindings are available by `(import (scheme show))`.

*scheme.show* is a combination of submodules `scheme.show.base`, `scheme.show.color`, `scheme.show.columnar` and `scheme.show.unicode`.

**scheme.show.base** [Module]
Exports bindings of R7RS `(scheme show base)` library. From R7RS programs, those bindings are available by `(import (scheme show base))`.

This contains most combinator formatting procedures.

**scheme.show.color** [Module]
Exports bindings of R7RS `(scheme show color)` library. From R7RS programs, those bindings are available by `(import (scheme show color))`.

This contains formatters to color text using ANSI escape codes.

**scheme.show.columnar** [Module]
Exports bindings of R7RS `(scheme show columnar)` library. From R7RS programs, those bindings are available by `(import (scheme show columnar))`.

This contains formatters to help format in columns.

**scheme.show.unicode** [Module]
Exports bindings of R7RS `(scheme show unicode)` library. From R7RS programs, those bindings are available by `(import (scheme show unicode))`.

#### Usage

Combinator formatting provides a functionality similar to `format` from SRFI-28. But instead of writing a template string, you can use S-expressions, which are called “formatters”. It’s also extensible.

The two main concepts in combinator formatting are formatters and states. Formatters are procedures that specify how or what you want to output. Formatters can be composed to produce complex format. Normal types are also accepted where a procedure takes a formatter, they are formatted with `displayed`.

Format states let us customize control formatting, for example how many precision digits, what character for padding . . .. Format states can be changed locally with `with` or `with!`.

The entry point to combinator formatting is `show`, which takes a sequence of formatters and outputs to a port or returns a string.
show output-dest fmt ...  
[Function]  
[R7RS show base] {scheme.show.base} This is the main entry for combinator formatting. All formatters are processed to produce a string to output-dest if it’s a port. If output-dest is #t, the current output port is used. If output-dest is #f, the output string is returned. show return value is otherwise undefined. Non-formatters are also accepted and will be wrapped in displayed formatter.

(show #f "π = " (with ((precision 2)) (acos -1)) nl)  
⇒ "π = 3.14\n"

Formatting Objects

displayed obj  
[Function]  
[R7RS show base] {scheme.show.base} The formatter that formats the object obj the same as display. This is the default formatter when you pass an object to show.

written obj  
[Function]  
[R7RS show base] {scheme.show.base} The formatter that formats the object obj the same as write. Formatting settings numeric and precision are respected for relevant number types as long as the result can still be passed to read.

written-simply obj  
[Function]  
[R7RS show base] {scheme.show.base} Similar to written but does not handle shared structures.

pretty obj  
[Function]  
[R7RS show base] {scheme.show.base} Pretty prints an object.

pretty-simply obj  
[Function]  
[R7RS show base] {scheme.show.base} Similar to pretty but does not handle shared structures.

escaped str [quote-ch esc-ch renamer]  
[Function]  
[R7RS show base] {scheme.show.base} Prints a string, adding esc-ch (#\ by default) in front of all quote-ch (#" by default).

If esc-ch is #f, escape all quote-ch by doubling it. If renamer is specified, it’s a procedure that takes one character and returns another character or #f. It serves two purposes: to allow quoting more than one character and to replace them with something else. If the procedure returns #f, the character in question is not escaped. Otherwise the character is escaped and replaced with a new one.

esc-ch could also be a string, but this is Gauche specific behavior.

(show #t (escaped "hello \"world\"")) ⇒ hello \"world\"
(show #t (escaped "hello \"world\"
\" #1)) ⇒ hel\lo "world"
(show #t (escaped "hello \"world\"
\" #1 \#x)) ⇒ hex\lxlo "wor\xlid"
(show #t (escaped "hello \"world\"
\#e #f
(lambda (x) (if (char=? x #\o) #\0 #f)))
⇒ heelleO "weOrld"

maybe-escaped str pred [quote-ch esc-ch renamer]  
[Function]  
[R7RS show base] {scheme.show.base} Determines if str needs to be escaped or not. If true, the string is wrapped with quote-ch. The original string is escaped with escaped.

The string needs to be escaped if any quote-ch or esc-ch is present, or any character that makes pred return #t.

(show #t (maybe-escaped "helloworld" char-whitespace?)) ⇒ helloworld
Formatting Numbers

**numeric num** [radix precision sign-rule comma-rule comma-sep decimal-sep]

[R7RS show base] \{scheme.show.base\} Formats a number. The default values are from state variables below.

(show #f (numeric 1000)) ⇒ "1000"
(show #f (numeric 1000 8)) ⇒ "1750"
(show #f (numeric 1000 8 2)) ⇒ "1750.00"
(show #f (numeric 1000 8 2 #t)) ⇒ "+1750.00"
(show #f (numeric -1000 8 2 (cons "(" ")"))) ⇒ "(1750.00)"
(show #f (numeric 1000 8 2 #t 2)) ⇒ "+17,50.00"
(show #f (numeric 1000 8 2 #t 2 #\')) ⇒ "+17'50.00"
(show #f (numeric 1000 8 2 #t 2 #\' #\:)) ⇒ "+17'50:00"

**numeric/comma num** [radix precision sign-rule]

[R7RS show base] \{scheme.show.base\} Formats a number with default comma-rule 3. See numeric for details.

(show #f (numeric/comma 1000)) ⇒ "1,000"
(show #f (with ((comma-sep #\.)) (numeric/comma 1000))) ⇒ "1.000"

**numeric/si num** [base separator]

[R7RS show base] \{scheme.show.base\} Formats a numeric with SI suffix. The default base is 1024 and uses suffix names like Ki, Mi, Gi... Other bases (e.g. 1000) use suffixes k, M, G... If separator is specified, it’s inserted between the number and suffix.

(show #f (numeric/si 1024)) ⇒ "1Ki"
(show #f (numeric/si 200000 1000)) ⇒ "200k"
(show #f (numeric/si 1024 1024 #\/)) ⇒ "1/Ki"

**numeric/fitted width n** [arg ...]

[R7RS show base] \{scheme.show.base\} Like numeric but if the result does not fit in width characters with current precision, outputs a string of hashes instead of the truncated and incorrect number.

(show #f (with ((precision 2)) (numeric/fitted 4 1.25))) ⇒ "1.25"
(show #f (with ((precision 2)) (numeric/fitted 4 12.345))) ⇒ ".#.##"

Formatting Space

**nl**

Outputs a newline.

(show #f nl) ⇒ "\n"

**fl**

Short for “fresh line”, makes sure the following output is at the beginning of the line.

(show #f fl) ⇒ 
(show #f "aaa" fl) ⇒ "aaa\n"

**nothing**

Outputs nothing. This is useful in combinators as default no-op in conditionals.
space-to column

Appends pad-char to reach the given column.

(show #f "abcdef" (space-to 3) "a") ⇒ "a"
(show #f "abcdef" (space-to 3) "a") ⇒ "abcdefa"

tab-to [tab-width]

Outputs pad-char to reach the next tab stop.

Concatenation

joined mapper list [sep]

Formats each element in list with mapper and inserts sep in between. sep by default is an empty string, but it could be any string or formatter.

(show #f (joined displayed (list "a" "b") "")) ⇒ "a b"
(show #f (joined displayed (list "a" "b") nl)) ⇒ "a\nb"

joined/suffix mapper list [sep]

Similar to joined except the separator is inserted after every element.

(show #f (joined/suffix displayed '1 2 3) nl)) ⇒ "1\n2\n3\n"

joined/last mapper last-mapper list [sep]

Similar to joined but last-mapper is used on the last element of list instead.

(show #f (joined/last displayed (lambda (last) (each "and " last))
'(lions tigers bears)
"", ")))
⇒ "lions, tigers, and bears"

joined/dot mapper dot-mapper list [sep]

Similar to joined but if list is a dotted list, then formats the dotted value with dot-mapper instead.

Padding and Trimming

padded width fmt...

Pads the output of fmt... on the left side with pad-char if it’s shorter than width characters.

(show #f (padded 10 "abc")) ⇒ "abc"
(show #f (with ((pad-char #\-)) (padded 10 "abc"))) ⇒ "-------abc"
padded/right width fmt ...  
[R7RS show base] {scheme.show.base} Similar to padded but except padding is on the right side instead.

padded/both width fmt ...  
[R7RS show base] {scheme.show.base} Similar to padded but except padding is on both sides, keeping the fmt output at the center.

trimmed width fmt ...  
[R7RS show base] {scheme.show.base} Trims the output of fmt... on the left so that the length is width characters or less. If ellipsis state variable is defined, it will be put on the left to denote trimming.

(show #f (trimmed 5 "hello world")) ⇒ "world"
(show #f (with ((ellipsis "..")) (trimmed 5 "hello world"))) ⇒ "..rld"

trimmed/right width fmt ...  
[R7RS show base] {scheme.show.base} Similar to trimmed but the trimming is on the right.

trimmed/both width fmt ...  
[R7RS show base] {scheme.show.base} Similar to trimmed but the trimming is on both sides, keeping the center of the output.

trimmed/lazy  width fmt ...  
[R7RS show base] {scheme.show.base} A variant of trimmed which generates each fmt in left to right order, and truncates and terminates immediately if more than width characters are generated. Thus this is safe to use with an infinite amount of output, e.g. from written-simply on an infinite list.

fitted width fmt ...  
[R7RS show base] {scheme.show.base} A combination of padded and trimmed, ensures the output width is exactly width, truncating if it goes over and padding if it goes under.

fitted/right width fmt ...  
[R7RS show base] {scheme.show.base} A combination of padded/right and trimmed/right, ensures the output width is exactly width, truncating if it goes over and padding if it goes under.

fitted/both width fmt ...  
[R7RS show base] {scheme.show.base} A combination of padded/both and trimmed/both, ensures the output width is exactly width, truncating if it goes over and padding if it goes under.

Columnar Formatting

columnar column ...  
[R7RS show columnar] {scheme.show.columnar}

tabular column ...  
[R7RS show columnar] {scheme.show.columnar}

wrapped fmt ...  
[R7RS show columnar] {scheme.show.columnar}

wrapped/list list-of-strings  
[R7RS show columnar] {scheme.show.columnar}
Chapter 10: Library modules - R7RS standard libraries

wrapped/char \textit{fmt} \ldots
\hfill [Function]
\hfill [R7RS show columnar] \{scheme.show.columnar\}

\textbf{justified} \textit{format} \ldots
\hfill [Function]
\hfill [R7RS show columnar] \{scheme.show.columnar\}

\textit{from-file} \textit{pathname}
\hfill [Function]
\hfill [R7RS show columnar] \{scheme.show.columnar\}

\textit{line-numbers} \textbf{[start]}
\hfill [Function]
\hfill [R7RS show columnar] \{scheme.show.columnar\}

\textbf{Colors}

as-red \textit{fmt} \ldots
\hfill [Function]
as-blue \textit{fmt} \ldots
\hfill [Function]
as-green \textit{fmt} \ldots
\hfill [Function]
as-cyan \textit{fmt} \ldots
\hfill [Function]
as-yellow \textit{fmt} \ldots
\hfill [Function]
as-magenta \textit{fmt} \ldots
\hfill [Function]
as-white \textit{fmt} \ldots
\hfill [Function]
as-black \textit{fmt} \ldots
\hfill [Function]
as-bold \textit{fmt} \ldots
\hfill [Function]
as-underline \textit{fmt} \ldots
\hfill [Function]
\hfill [R7RS show color] \{scheme.show.color\} Outputs the ANSI escape code to make all \textit{fmt} \ldots a given color or style.

\textbf{Unicode}

as-unicode \textit{fmt} \ldots
\hfill [Function]
\hfill [R7RS show unicode] \{scheme.show.unicode\}

\textit{unicode-terminal-width} \textit{str}
\hfill [Function]
\hfill [R7RS show unicode] \{scheme.show.unicode\}

\textbf{Higher Order Formatters and State}

\textit{fn} \((\text{\textit{id state-var} \ldots}) \textit{expr} \ldots \textit{fmt})
\hfill [Function]
\hfill [R7RS show base] \{scheme.show.base\} This is short for “function” and the analog to \texttt{lambda}. It returns a formatter which on application evaluates each \textit{expr} and \textit{fmt} in left-to-right order, in a lexical environment extended with each identifier \textit{id} bound to the current value of the state variable named by the symbol \textit{state-var}. The result of the \textit{fmt} is then applied as a formatter.

As a convenience, any \((\textit{id state-var} \ldots)\) list may be abbreviated as simply \textit{id}, indicating \textit{id} is bound to the state variable of the same (symbol) name.

\textit{with} \((\text{\textit{state-var value} \ldots}) \textit{fmt} \ldots)
\hfill [Function]
\hfill [R7RS show base] \{scheme.show.base\} This is the analog of \texttt{let}. It temporarily binds specified state variables with new values for \textit{fmt} \ldots.

\textit{with!} \textit{state-var value} \textit{fmt} \ldots
\hfill [Function]
\hfill [R7RS show base] \{scheme.show.base\} Similar to \textit{with} but the value updates persist even after \textit{with!}. 
for kek fmt1 fmt2

[Function]
[R7RS show base] {scheme.show.base} Calls fmt1 on (a conceptual copy of) the current state, then fmt2 on the same original state as though fmt1 had not been called (i.e. any potential state mutation by fmt1 does not affect fmt2).

call-with-output formsr mapper

[Function]
[R7RS show base] {scheme.show.base} A utility, calls formatter on a copy of the current state (as with forked), accumulating the results into a string. Then calls the formatter resulting from (mapper result-string) on the original state.

Standard State Variables

port

The current port output is written into, could be overriden to capture intermediate output.

row

The current row of the output.

col

The current column of the output.

width

The current line width, used for wrapping, pretty printing and columnar formatting.

output

The underlying standard formatter for writing a single string.

writer

The mapper for automatic formatting of non-string/char values in top-level show, each and other formatters. Default value is implementation-defined.

string-width

A function of a single string. It returns the length in columns of that string, used by the default output.

pad-char

The character used for by padding formatters, #\space by default

ellipses

The string used when truncating as described in trimmed.

radix

The radix for numeric output, 10 by default. Valid values are from 2 to 36.

precision

The number of digits written after the decimal point for numeric output. The value is rounded if the numeric value written out requires more digits than requested precision. See the SRFI for exact rounding behavior.

sign-rule

If #t, always output the plus sign + for positive numbers. If sign-rule is a par of two strings, negative numbers are printed with the strings wrapped around (and no preceding negative sign -).
**comma-rule**

The number of digits between commas, specified by `comma-sep`.

**comma-sep**

The character used as comma for numeric formatting, `\#`, by default.

**decimal-sep**

The character to use for decimals in numeric formatting. The default depends on `comma-sep`, if it’s `\#`, then the decimal separator is `\#`, otherwise it’s `\#`.

**decimal-align**

[Variable]

**word-separator?**

[Variable]
11 Library modules - SRFIs

This chapter lists modules that provides SRFI functionalities. Note that some of SRFI features are built in Gauche core and not listed here. The SRFIs that have been adopted to R7RS-large are described in Chapter 10 [Library modules - R7RS standard libraries], page 580.

See Section 2.1 [Standard conformance], page 5, for entire list of supported SRFIs.

(Even if a srfi is not listed here, you can still say (use srfi.N) or (import (srfi N)), as far as srfi N is supported by Gauche.)

Note that we use a hyphen to refer to a SRFI as a specification (e.g. SRFI-13) but use a hierarchical name for the module (e.g. srfi.13 as a Gauche’s module, or (srfi 13) as a R7RS library).

11.1 Testing availability of SRFIs

To test availability of a SRFI with cond-expand (see Section 4.12 [Feature conditional], page 75), you can use (library library-name) feature requirement. You can use both Gauche-style and R7RS-style library name notation.

;;; Use 'library' feature requirement with Gauche module name
(cond-expand
  [(library srfi.13) (use srfi.13) ...]
  [else ...])

;;; Use 'library' feature requirement with R7RS library name
(cond-expand
  [(library (srfi 13)) (import (srfi 13)) ...]
  [else ...])

Alternatively you can use a SRFI name as a feature identifier. This use is deprecated. This is an old style used before R7RS. With this style, if the feature is fulfilled, the necessary libraries are implicitly imported, so you don’t need to have use or import forms (but that means you can’t control the imports with only, rename, prefix etc.)

;;; DEPRECATED. Use 'srfi-N' feature identifier.
;;; If the feature is fulfilled, you can assume the module is already 'use'd.
(cond-expand
  [srfi-13 ...]
  [else ...])

If srfi-N is used as a feature identifier, a warning is issued when an environment variable GAUCHE_WARN_SRFI_FEATURE_ID is set. In future this warning will be the default. We urge the users to update their code to use (library srfi.N) form.

11.2 SRFIs that have become R7RS-large

Here’s a list of SRFIs that have become a part of R7RS-large, ordered by SRFI numbers.

SRFI-1 List library
    scheme.list (see Section 10.3.1 [R7RS lists], page 593).

SRFI-14 Character-set library
    scheme.char-set (see Section 10.3.6 [R7RS character sets], page 614).

SRFI-41 Streams
    scheme.stream (see Section 10.3.14 [R7RS stream], page 635).

SRFI-101 Purely functional random-access pairs and lists
    scheme.rlist (see Section 10.3.9 [R7RS random-access lists], page 622).
Chapter 11: Library modules - SRFIs

SRFI-111 Boxes
   \texttt{scheme.box} (see Section 10.3.15 [R7RS boxes], page 636).

SRFI-113 Sets and bags
   \texttt{scheme.set} (see Section 10.3.5 [R7RS sets], page 606).

SRFI-115 Scheme Regular Expressions
   \texttt{scheme.regex} (see Section 10.3.19 [R7RS regular expressions], page 640).

SRFI-116 Immutable list library
   \texttt{scheme.\texttt{\texttt{i}}list} (see Section 10.3.8 [R7RS immutable lists], page 621).

SRFI-117 Queues based on lists
   \texttt{scheme.list-queue} (see Section 10.3.16 [R7RS list queues], page 636).

SRFI-121 Generators
   \texttt{scheme.generator} (see Section 10.3.12 [R7RS generators], page 631).

SRFI-124 Ephemerons
   \texttt{scheme.ephemeron} (see Section 10.3.17 [R7RS ephemerons], page 639).

SRFI-125 Intermediate hash tables
   \texttt{scheme.hash-table} (see Section 10.3.7 [R7RS hash tables], page 618).

SRFI-127 Lazy sequences
   \texttt{scheme.lseq} (see Section 10.3.13 [R7RS lazy sequences], page 633).

SRFI-128 Comparators (reduced)
   \texttt{scheme.comparator} (see Section 10.3.18 [R7RS comparators], page 640).

SRFI-132 Sort library
   \texttt{scheme.sort} (see Section 10.3.4 [R7RS sort], page 602).

SRFI-133 Vector library
   \texttt{scheme.vector} (see Section 10.3.2 [R7RS vectors], page 597).

SRFI-134 Immutable deques
   \texttt{scheme.ideque} (see Section 10.3.10 [R7RS immutable deques], page 623).

SRFI-135 Immutable texts
   \texttt{scheme.text} (see Section 10.3.11 [R7RS immutable texts], page 627).

SRFI-141 Integer division
   \texttt{scheme.division} (see Section 10.3.21 [R7RS integer division], page 663).

SRFI-143 Fixnums
   \texttt{scheme.fixnum} (see Section 10.3.23 [R7RS fixnum], page 668).

SRFI-144 Flonums
   \texttt{scheme.flonum} (see Section 10.3.24 [R7RS flonum], page 670).

SRFI-146 Mappings
   \texttt{scheme.mapping}, \texttt{scheme.mapping.hash} (see Section 10.3.20 [R7RS mappings], page 652).

SRFI-151 Bitwise operations
   \texttt{scheme.bitwise} (see Section 10.3.22 [R7RS bitwise operations], page 664).

SRFI-158 Generators and accumulators
   \texttt{scheme.generator} (see Section 10.3.12 [R7RS generators], page 631).

SRFI-159 Combinator formatting
   \texttt{scheme.show} (see Section 10.3.26 [R7RS combinator formatting], page 681).

SRFI-160 Homogeneous numeric vector libraries
   \texttt{scheme.vector.@} (see Section 10.3.3 [R7RS uniform vectors], page 602).
11.3 srfi.4 - Homogeneous vectors

srfi.4 [Module]
SRFI-4 is now implemented in gauche.uvector module See Section 6.13.2 [Uniform vectors], page 198. This module simply inherits gauche.uvector for backward-compatibility.

11.4 srfi.5 - A compatible let form with signatures and rest arguments

srfi.5 [Module]
This module provides SRFI-5’s extended let syntax.

let ((var val) ... [. (rest val ...)]) body ... [Macro]
let (name ((var val) ... [. (rest val ...)]) body ...) [Macro]

[SRFI-5]{srfi.5} The let syntax is extended in two ways.
- The extended let syntax accepts the name identifier (for named let syntax) within the list of bindings (as in the third syntax above).
- The extended let syntax accepts the rest parameter binding which works like the rest parameter in the lambda syntax.

See SRFI-5 document for rationale of this extension.

11.5 srfi.7 - Feature-based program configuration language

srfi.7 [Module]
This module provides a program configuration metalanguage (program form) defined in srfi-7. Gauche autoloads srfi.7 module, so you don’t need to say (use srfi.7) explicitly. Note that the program form isn’t necessary to be a Scheme expression. SRFI-7 allows an implementation to preprocess the program form to produce a Scheme program, then executes it with different means. Gauche implements program form as a macro, so it can evaluates the form directly. Nonetheless, it doesn’t make sense to mix program form and other forms in one file, or expecting a return value of program form. A typical usage of program form is to prepare a single file which just contains program form. (It can load other files using files clause (see below) within the program form.) To execute such a program file in Gauche, you can just load it.

program program-clause program-clause2 ... [Configuration Language]
[SRFI-7]{srfi.7} This is a configuration language to structure a Scheme program, based on availability of the features.
A Scheme program is constructed from the program form. Gauche evaluates the constructed Scheme program on-the-fly.
Each program-clause needs to be one of the "Program Clauses" below.

requires feature-id feature-id2 ... [Program Clause]
[SRFI-7] The feature-id’s are the same as SRFI-0’s (see Section 4.12 [Feature conditional], page 75). It tells that the following code requires these feature-id’s.
If a feature-id which is not supported in Gauche is given, an error is signaled.

files filename ... [Program Clause]
[SRFI-7] Inserts the content of the filenames into a program. In Gauche, this clause just causes filenames to be loaded into the current module.
code scheme-expression . . .  
[SRFI-7] The scheme-expressions are inserted into a program.

feature-cond clause clause2 . . .  
[SRFI-7] Clause is a following form:

(requirement program-clause program-clause2 . . .)

Where requirement should be one of the following:
- feature-id
- (and requirement . . .)
- (or requirement . . .)
- (not requirement)

The requirement of the last clause may be else.
Gauche checks each requirement one by one, and if it finds a fulfilled requirement, inserts the program-clauses in that clause into the program.

11.6 srfi.13 - String library

srfi.13  
[Module]
Defines a large set of string-related procedures.

It was one of the earliest popular srfis, but some of the procedures were turned out not to align well with recent Scheme developments. See Section 11.29 [String library (reduced)], page 740, and Section 11.28 [Cursor-based string library], page 738, adapts this srfis to more “modern” form. Consider to use them for newer development.

Many procedures in this srfi specifies the position of a character in a string with an integer index. Although it’s portable, it is not optimal for multibyte strings. Gauche natively supports string cursors, which is more efficient than integer indexes (see Section 6.11.5 [String cursors], page 174). All SRFI-13 procedures that accepts integer indexes can also accept string cursors.

11.6.1 General conventions

There are a few common factors in string library API, which I don’t repeat in each function description

argument convention

The following argument names imply their types.

s, s1, s2  Those arguments must be strings.

char/char-set/pred

This argument can be a character, a character-set object, or a predicate that takes a single character and returns a boolean value. “Applying char/char-set/pred to a character” means, if char/char-set/pred is a character, it is compared to the given character; if char/char-set/pred is a character set, it is checked if the character set contains the given character; if char/char-set/pred is a procedure, it is applied to the given character. “A character satisfies char/char-set/pred” means such application to the character yields true value.

start, end  Lots of SRFI-13 functions takes these two optional arguments, which limit the area of input string from start-th character (inclusive) to end-th character (exclusive), where the operation is performed. When specified, the condition 0 <= start <= end <= length of the string must
be satisfied. Default value of start and end is 0 and the length of the string, respectively.

**shared variant**

Some functions have variants with “/shared” attached to its name. SRFI-13 defines those functions to allow to share the part of input string, for better performance. Gauche doesn’t have a concept of shared string, and these functions are mere synonyms of their non-shared variants. However, Gauche internally shares the storage of strings, so generally you don’t need to worry about the overhead of copying substrings.

**right variant**

Most functions works from left to right of the input string. Some functions have variants with “-right” to its name, that works from right to left.

### 11.6.2 String predicates

**string-null? s**  
[Function]  
[SRFI-13]{srfi.13} Returns #t if s is an empty string, "".

**string-every char/char-set/pred s :optional start end**  
[Function]  
[SRFI-13]{srfi.13} Sees if every character in s satisfies char/char-set/pred. If so, string-every returns the value that is returned at the last application of char/char-set/pred. If any of the application returns #f, string-every returns #f immediately.

**string-any char/char-set/pred s :optional start end**  
[Function]  
[SRFI-13]{srfi.13} Sees if any character in s satisfies char/char-set/pred. If so, string-any returns the value that is returned by the application. If no character satisfies char/char-set/pred, #f is returned.

### 11.6.3 String constructors

**string-tabulate proc len**  
[Function]  
[SRFI-13]{srfi.13} proc must be a procedure that takes an integer argument and returns a character. string-tabulate creates a string, whose i-th character is calculated by (proc i).

```
(string-tabulate
 (lambda (i) (integer->char (+ i #x30))) 10)
⇒ "0123456789"
```

**string-unfold p f g seed :optional base make-final**  
[Function]  
[SRFI-13]{srfi.13} A fundamental string builder. The p, f and g are procedures, taking the current seed value. The stop predicate p determines when to stop: If it returns a true value, string building stops. The mapping function f returns a character from the current seed value. The next seed function g returns a next seed value from the current seed value. The seed argument gives the initial seed value.

```
(string-unfold (\n (= n 10))
 (\n (integer->char (+ n 48)))
 (\n (+ n 1))
 0)
⇒ "0123456789"
```

The optional argument base is, when given, prepended to the result string. Another optional argument make-final is a procedure that takes the last return value of g and returns a string that becomes the suffix of the result string.
(string-unfold (\n (= n 10))
  (\n (integer->char (+ n 48)))
  (\n (+ n 1))
  0 "foo" x->string)
⇒ "foo012345678910"

**string-unfold-right**  \( p \ f \ g \ seed :optional \ base \ make-final \)  

[SRFI-13]\{srfi.13\} Another fundamental string builder. The meanings of arguments are the same as **string-unfold**. The only difference is that the string is build right-to-left. The optional **base**, if given, becomes the suffix of result, and the result of **make-final** becomes the prefix.

(string-unfold-right (\n (= n 10))
  (\n (integer->char (+ n 48)))
  (\n (+ n 1))
  0 "foo" x->string)
⇒ "109876543210foo"

**reverse-list->string**  \( char-list \)  

[SRFI-13]\{srfi.13\} \(=\) `(list->string (reverse char-list))

### 11.6.4 String selection

**substring/shared**  \( s \ start :optional \ end \)  

[SRFI-13]\{srfi.13\} In Gauche, this is the same as **substring**, except that the **end** argument is optional.

(substring/shared "abcde" 2) ⇒ "cde"

**string-copy!**  \( target \ tstart \ s :optional \ start \ end \)  

[SRFI-13]\{srfi.13\} Copies a string \( s \) into a string **target** from the position **tstart**. The **target** string must be mutable. Optional **start** and **end** arguments limits the range of \( s \). If the copied string run over the end of **target**, an error is signaled.

(define s (string-copy "abcde"))
(string-copy! s 2 "ZZ")
s ⇒ "abZZe"

It is ok to pass the same string to **target** and **s**; this always work even if the regions of source and destination are overlapping.

Note that Gauche encourages you to treat strings as immutable objects. Internally, a string is an indirect pointer to a immutable entity, and mutating a string means copying the original entity and creating a new one. It doesn’t “save allocations”. Always use the functional version **string-copy** unless you absolutely need to replace a string in-place. See Section 6.11.9 [String utilities], page 178.

**string-take**  \( s \ nchars \)  

**string-drop**  \( s \ nchars \)  

**string-take-right**  \( s \ nchars \)  

**string-drop-right**  \( s \ nchars \)  

[SRFI-13]\{srfi.13\} Returns the first \( nchars \)-character string of \( s \) (**string-take** or the string without first \( nchars \) (**string-drop**). The **-right** variation counts from the end of string. It is guaranteed that the returned string is always a copy of \( s \), even no character is dropped.

(string-take "abcde" 2) ⇒ "ab"
(string-drop "abcde" 2) ⇒ "cde"
(string-take-right "abcde" 2) ⇒ "de"
(string-drop-right "abcde" 2) ⇒ "abc"

string-pad s len :optional char start end
[string function]
[SRFI-13]{srfi.13} If a string s is shorter than len, returns a string of len where char is padded to the left or right, respectively. If s is longer than len, the rightmost or leftmost len chars are taken. Char defaults to #\space. If start and end are provided, the substring of s is used as the source.

(string-pad "abc" 10) ⇒ " abc"
(string-pad "abcdefg" 3) ⇒ "efg"

(string-pad-right "abc" 10) ⇒ "abc 

(string-pad "abcdefg" 10 #\+ 2 5) ⇒ "+++++++cde"

string-trim s :optional char/char-set/pred start end
[string function]
string-trim-right s :optional char/char-set/pred start end
[string function]
string-trim-both s :optional char/char-set/pred start end
[string function]
[SRFI-13]{srfi.13} Removes characters that match char/char-set/pred from s. String-trim removes the characters from left of s, string-trim-right does from right, and string-trim-both does from both sides. Char/char-set/pred defaults to #\[\], i.e. a char-set of whitespaces. If start and end are provided, the substring of s is used as the source.

(string-trim " abc ") ⇒ "abc 
(string-trim-right " abc ") ⇒ " abc"
(string-trim-both " abc ") ⇒ "abc"

11.6.5 String comparison

string-compare s1 s2 proc< proc= proc> :optional start1 end1 start2 end2
[string function]
string-compare-ci s1 s2 proc< proc= proc> :optional start1 end1 start2 end2
[string function]
[SRFI-13]{srfi.13} Compares two strings s1 and s2 codepoint-wise from left. When mismatch is found at the index k of s1, calls proc< with k if s1’s codepoint is smaller than the corresponding s2’s, or calls proc> if s1’s one is greater than s2’s. If two strings are the same, calls proc= with the index of the last compared position in s1.

(string-compare "abcd" "abzd"
  (^i '(< ,i)) (^i ' (= ,i)) (^i ' (> ,i)))
⇒ (< 2)

(string-compare "abcd" "abcd"
  (^i '(< ,i)) (^i ' (= ,i)) (^i ' (> ,i)))
⇒ (= 3)

The optional arguments restricts the range of the input strings; however, the index passed to one of the procedures is always an index from the beginning of s1.

(string-compare "zzabcdyy" "abcz"
  (^i '(< ,i)) (^i ' (= ,i)) (^i ' (> ,i)) 2 6 0 4)
⇒ (< 5)
(string-compare "zzabcdyy" "abcz"
 (`i '<(i)) (`i '='(i)) (`i '>(i)) (2 5 0 3))
⇒ (= 4)

The case-insensitive variant, string-compare-ci, compares each codepoint with character-wise case-folding. It won’t consider special case folding such as German eszett.

### string= s1 s2 :optional start1 end1 start2 end2

[Function]

### string<> s1 s2 :optional start1 end1 start2 end2

[Function]

### string< s1 s2 :optional start1 end1 start2 end2

[Function]

### string<= s1 s2 :optional start1 end1 start2 end2

[Function]

### string> s1 s2 :optional start1 end1 start2 end2

[Function]

### string>= s1 s2 :optional start1 end1 start2 end2

[Function]

[SRFI-13]{srfi.13} Compare two strings s1 and s2. Optional arguments can limit the portion of strings to be compared. Comparison is done by character-wise.

Note: The builtin procedures string=? etc. can also be used for character-wise string comparison, but they take arguments differently. See Section 6.11.8 [String comparison], page 177.

### string-ci= s1 s2 :optional start1 end1 start2 end2

[Function]

### string-ci<> s1 s2 :optional start1 end1 start2 end2

[Function]

### string-ci< s1 s2 :optional start1 end1 start2 end2

[Function]

### string-ci<= s1 s2 :optional start1 end1 start2 end2

[Function]

### string-ci> s1 s2 :optional start1 end1 start2 end2

[Function]

### string-ci=> s1 s2 :optional start1 end1 start2 end2

[Function]

[SRFI-13]{srfi.13} Compare two strings s1 and s2 in case-insensitive way. Optional arguments can limit the portion of strings to be compared. Case folding and comparison is done by character-wise, so they don’t consider case folding that affects multiple characters.

Note: We have two other sets of string comparison operations, both are named as string-ci=? etc. The builtin version (see Section 6.11.8 [String comparison], page 177) does character-wise comparison. The one in gauche.unicode uses full-string case conversion (see Section 9.37.3 [Full string case conversion], page 554). R7RS version is the latter.

### string-hash s :optional bound start end

[Function]

### string-ci-hash s :optional bound start end

[Function]

[SRFI-13]{srfi.13} (Note: Gauche has builtin string-hash and string-ci-hash according to SRFI-128. See Section 6.2.3 [Hashing], page 113, for the details. SRFI-13’s API is upper-compatible to SRFI-128’s. The underlying hash algorithm is the same as the builtin ones, so string-hash returns the same value as the builtin ones for the same string if optional arguments are omitted. On the other hand, the builtin string-ci-hash uses string case folding (e.g. German eszett and SS are the same), while SRFI-13’s string-hash-ci uses character-wise case folding. Unless there’s a strong reason, we recommend new code should use builtin SRFI-128 version instead of SRFI-13.)

Calculates hash value of a string s. For string-hash-ci, character-wise case folding is done before calculating the hash value.

If the optional bound argument is given, it must be a positive exact integer, and the return value is limited below it. The optional start and end arguments allows using that portion for calculation.

### 11.6.6 String prefixes & suffixes

### string-prefix-length s1 s2 :optional start1 end1 start2 end2

[Function]

### string-suffix-length s1 s2 :optional start1 end1 start2 end2

[Function]
Chapter 11: Library modules - SRFIs

string-prefix-length-ci s1 s2 :optional start1 end1 start2 end2

string-suffix-length-ci s1 s2 :optional start1 end1 start2 end2

| [SRFI-13] | {srfi.13} | Returns the length of the longest common prefix/suffix of two strings, s1 and s2. The optional arguments restrict the range of search. The *-ci variations use case folding character comparison.

(string-prefix-length "abacus" "abalone") ⇒ 3
(string-prefix-length "machine" "umbrella") ⇒ 0
(string-suffix-length "peeking" "poking") ⇒ 4

(string-prefix-length "obvious" "oblivious" 2 7 4 9) ⇒ 5

string-prefix? s1 s2 :optional start1 end1 start2 end2

string-suffix? s1 s2 :optional start1 end1 start2 end2

| [SRFI-13] | {srfi.13} | Returns true iff s1 is a prefix or suffix of s2, respectively. The optional arguments limit the range of s1 and s2 to look at. The *-ci variations use case folding character comparison.

(string-prefix? "sch" "scheme") ⇒ #t
(string-prefix? "lisp" "scheme") ⇒ #f

(string-suffix? "eme" "scheme") ⇒ #t
(string-suffix? "eme" "lisp") ⇒ #f

(string-prefix? "mit-scheme" "scheme-family" 4) ⇒ #t

11.6.7 String searching

string-index s char/char-set/pred :optional start end

string-index-right s char/char-set/pred :optional start end

| [SRFI-13] | {srfi.13} | Looks for the first element in a string s that matches char/char-set/pred, and returns its index. If char/char-set/pred is not found in s, returns #f. Optional start and end limit the range of s to search.

(string-index "Aloha oe" #\a) ⇒ 4
(string-index "Aloha oe" #\[Aa]) ⇒ 0
(string-index "Aloha oe" #\[s]) ⇒ 5
(string-index "Aloha oe" char-lower-case?) ⇒ 1
(string-index "Aloha oe" #\o 3) ⇒ 6

See also the Gauche built-in procedure string-scan (Section 6.11.9 [String utilities], page 178), if you need speed over portability.

string-skip s char/char-set/pred :optional start end

string-skip-right s char/char-set/pred :optional start end

| [SRFI-13] | {srfi.13} | Looks for the first element that does not match char/char-set/pred and returns its index. If such element is not found, returns #f. Optional start and end limit the range of s to search.

string-count s char/char-set/pred :optional start end

| [SRFI-13] | {srfi.13} | Counts the number of elements in s that matches char/char-set/pred. Optional start and end limit the range of s to search.
Chapter 11: Library modules - SRFIs

string-contains s1 s2 :optional start1 end1 start2 end2
[Function]

string-contains-ci s1 s2 :optional start1 end1 start2 end2
[Function]
[SRFI-13]{srfi.13} Looks for a string s2 inside another string s1. If found, returns an index in s1 from where the matching string begins. Returns #f otherwise. Optional start1, end1, start2 and end2 limits the range of s1 and s2.

See also the Gauche built-in procedure string-scan (Section 6.11.9 [String utilities], page 178), if you need speed over portability.

11.6.8 String case mapping

string-titlecase s :optional start end
[Function]

string-titlecase! s :optional start end
[Function]

string-upcase s :optional start end
[Function]

string-upcase! s :optional start end
[Function]

string-downcase s :optional start end
[Function]

string-downcase! s :optional start end
[Function]
[SRFI-13]{srfi.13} Converts a string s to titlecase, upcase or downcase, respectively. These operations uses character-by-character mapping provided by char-upcase etc. That is, string-upcase and string-downcase can be understood as follow:

\[(\text{string-upcase } s) \equiv (\text{string-map char-upcase } s)\]
\[(\text{string-downcase } s) \equiv (\text{string-map char-downcase } s)\]

If you need full case mapping that handles the case when a character is mapped to more than one characters, use the procedures with the same name in gauche.unicode module (see Section 9.37.3 [Full string case conversion], page 554).

The linear-update version string-titlecase!, string-upcase! and string-downcase! destroys s to store the result. Note that in Gauche, using those procedures doesn’t save anything, since string mutation is expensive by design. They are provided merely for completeness.

11.6.9 String reverse & append

string-reverse s :optional start end
[Function]

string-reverse! s :optional start end
[Function]
[SRFI-13]{srfi.13} Returns a string in which the character positions are reversed from s. string-reverse! modifies s.

\[(\text{string-reverse } "mahalo") \Rightarrow "olaham"\]
\[(\text{string-reverse } "mahalo" 3) \Rightarrow "ola"\]
\[(\text{string-reverse } "mahalo" 1 4) \Rightarrow "aha"\]

\((\text{let } ((s (\text{string-copy } "mahalo"))) \text{(string-reverse! } s 1 5) s) \Rightarrow "mlahao"\)

string-concatenate string-list
[Function]
[SRFI-13]{srfi.13} Concatenates list of strings.

\[(\text{string-concatenate } '("humuhumu" "nukunuku" "apua" "ʻa")) \Rightarrow "humuhumunukunukuapuaʻa"\]
string-concatenate/shared string-list

string-append/shared s ...

[SRFI-13]{srfi.13} “Shared” version of string-concatenate and string-append. In Gauche, these are just synonyms of them.

string-concatenate-reverse string-list

string-concatenate-reverse/shared string-list

[SRFI-13]{srfi.13} Reverses string-list before concatenation. “Shared” version works the same in Gauche.

11.6.10 String mapping

string-map! proc s :optional start end

[SRFI-13]{srfi.13} string-map! applies proc on every character of s, and stores the results into s. It is an error if proc returns non-character.

(let ((s (string-copy "wikiwiki")))
  (string-map! char-upcase s 4)
  s)
⇒ "wikiWIKI"

string-fold kons knil s :optional start end

string-fold-right kons knil s :optional start end

[SRFI-13]{srfi.13} Like fold and fold-right (see Section 6.6.6 [Walking over lists], page 147), but works on a string instead of a list.

(string-fold cons '() "abcde")
⇒ (#\e #\d #\c #\b #\a)
(string-fold-right cons '() "abcde")
⇒ (#\a #\b #\c #\d #\e)

string-for-each-index proc s :optional start end

[SRFI-13]{srfi.13} Call proc on each index of the string s, from left to right. The result of proc is discarded. The optional start and end arguments can be an integer index or a string cursor, restricting the range of the input string to be traversed.

11.6.11 String rotation

xsubstring s from :optional to start end

[SRFI-13]{srfi.13} Takes a substring of infinite repetition of string s between index from (inclusive) and index to (exclusive). If to is omitted, the length of s is assumed.

For example, if s is "abcde", we repeat it infinitely to both sides. So \(n\)-th character for integer \(n\) is always \#\a, which extends negative \(n\) as well.

(xsubstring "abcde" 2 10)
⇒ "cdeabcde"
(xsubstring "abcde" -9 -2)
⇒ "bcdeabc"

The optional start and end arguments can be an integer index or a string cursor, it works as (xsubstring (substring s start end) from to).

string-xcopy! target tstart s sf from :optional sto start end

[SRFI-13]{srfi.13} It works as (string-copy! target tstart (xsubstring s sf from sto start end)).
Chapter 11: Library modules - SRFIs

11.6.12 Other string operations

string-replace s1 s2 start1 end1 :optional start2 end2  [Function]
[SRFI-13]{srfi.13} Returns a new string whose content is a copy of a string s1, except the part beginning from the index start1 (inclusive) and ending at the index end1 (exclusive) is replaced by a string s2. When optional start2 and end2 arguments are given, s2 is trimmed first according to them. The size of the gap, (- end1 start1), doesn’t need to be the same as the size of the inserted string. Effectively, this is the same as the following code.

\[
\text{(string-append (substring s1 0 start1) (substring s2 start2 end2) (substring s1 end1 (string-length s1)))}
\]

string-tokenize s :optional token-set start end  [Function]
[SRFI-13]{srfi.13} Splits the string s into a list of substrings, where each substring is a maximal non-empty contiguous sequence of characters from the character set token-set. The default of token-set is char-set:graphic (see Section 6.10.2 [Predefined character sets], page 167).

See also Gauche’s built-in string-split (see Section 6.11.9 [String utilities], page 178), which provides similar features but different criteria.

11.6.13 String filtering

string-filter char/char-set/pred s :optional start end  [Function]
string-delete char/char-set/pred s :optional start end  [Function]
[SRFI-13]{srfi.13} Returns a string consists of characters in a string s that passes (or don’t pass) the test indicated by char/char-set/pred, respectively.

\[
\text{(string-filter char-upper-case? "Hello, World!")}
\Rightarrow "HW"
\]

\[
\text{(string-delete char-upper-case? "Hello, World!")}
\Rightarrow "ello, orld!"
\]

\[
\text{(string-delete #\l "Hello, World!")}
\Rightarrow "Heo, Word!"
\]

\[
\text{(string-filter #\[w] "Hello, World!")}
\Rightarrow "HelloWorld"
\]

Note: SRFI-13 was revised after finalization to switch the order of arguments char/char-set/pred and s was. At the time of finalization, the order was (string-filter s pred) and Gauche implemented it accordingly. However, most existing implementations follows the revised order, since that was what the SRFI-13 reference implementation had.

So, from 0.9.4, we revised the API to comply the current SRFI-13 spec, but we also accept the old order as well not to break the old code. We recommend the new code to use the new order.

11.6.14 Low-level string procedures

Here are some helper procedures useful to write other string-processing utilities similar to SRFI-13:
string-parse-start+end proc s args  [Function]
string-parse-final-start+end proc s args  [Function]
[SRFI-13]{srfi.13} Most SRFI-13 procedures takes optional start and end arguments. These
procedures looks for them in the rest arguments args, and if they’re not provided, gives the
default values.

The proc argument is the name (symbol) of the procedure, to be used in the error condition,
and s is the string to be processed.

Both expects an optional start argument at the beginning of args, followed by an optional end
argument. If the end argument is missing, the length of s is assumed. If the start argument
is also missing, 0 is assumed. If arguments are provided, they must be exact integers, and 0
<= start <= end <= (string-length s) must be satisfied; otherwise, an error is signaled.

If more than two arguments are in args, string-parse-final-start+end raises an error (in
other words, it requires that the argument list end with the end argument at most), while
string-parse-start+end permits it and returns the result of arguments, along with start and
end indexes.

Both function return three values: The rest of the argument list, the start index, and the end
index.

let-string-start+end (start end [rest]) proc-exp s-exp args-exp body ...  [Macro]
[SRFI-13]{srfi.13}

check-substring-spec proc s start end  [Function]
substring-spec-ok? s start end  [Function]
[SRFI-13]{srfi.13}

make-kmp-restart-vector s :optional c= start end  [Function]
[SRFI-13]{srfi.13}

kmp-step pat rv c i c= p-start  [Function]
[SRFI-13]{srfi.13}

string-kmp-partial-search pat rv s i :optional c= p-start s-start s-end  [Function]
[SRFI-13]{srfi.13}

11.7 srfi.19 - Time data types and procedures

srfi.19  [Module]
This SRFI defines various representations of time and date, and conversion methods among
them.

On Gauche, time object is supported natively by <time> class (see Section 6.24.9 [Time],
page 316). Date object is supported by <date> class described below.

11.7.1 Time types
Time type is represented by a symbol. This module defines the following constant variables that
is bound to its name, for convenience.

time-utc  [Constant]
[SRFI-19]{srfi.19} UTC time. Gauche’s built-in current-time always returns this type
(see Section 6.24.9 [Time], page 316).

time-tai  [Constant]
[SRFI-19]{srfi.19} International Atomic Time. This time is a bit larger than UTC, due to
the leap seconds.
Chapter 11: Library modules - SRFIs

11.7.1 SRFI-19 Time types

- **time-monotonic**
  [SRFI-19]{srfi.19} Implementation-dependent monotonically increasing time. In Gauche, this is the same as time-tai.

- **time-duration**
  [SRFI-19]{srfi.19} Duration between two absolute time points.

- **time-process**
  [SRFI-19]{srfi.19} CPU time in current process. Gauche calculates this from user time and system time returned by POSIX times(3).

- **time-thread**
  [SRFI-19]{srfi.19} CPU time in current thread. In the current implementation, this is the same as time-process.

11.7.2 Time queries

- **current-time**: optional time-type
  [SRFI-19]{srfi.19} Extends Gauche built-in current-time (see Section 6.24.9 [Time], page 316) to take optional time-type argument to specify the desired time type. time-type must be one of the types described in Section 11.7.1 [SRFI-19 Time types], page 701.

- **current-date**: optional tz-offset
  [SRFI-19]{srfi.19} Returns the current date as an instance of <date> class (see Section 11.7.4 [SRFI-19 Date], page 703). If tz-offset is given, it must be an offset from UTC in number of seconds. If tz-offset is not given, returns the date in local time zone.

- **current-julian-day**
  [SRFI-19]{srfi.19} Returns the current julian day, a point in time as a real number of days since -4714-11-24T12:00:00Z (November 24, -4714 at noon, UTC).

- **current-modified-julian-day**
  [SRFI-19]{srfi.19} Returns the current modified julian day, a point in time as a real number of days since 1858-11-17T00:00:00Z (November 17, 1858 at midnight, UTC).

- **time-resolution**: optional type
  [SRFI-19]{srfi.19} Returns clock resolution of the time type type in nanoseconds, as an exact positive integer. If type is omitted, time-utc is assumed.
  
  Note: In the current implementation, the return value isn’t actual resolution, but a value that’s guaranteed to be equal to or greater than the actual resolution.

11.7.3 Time procedures

- **make-time** type nanoseconds seconds
  [SRFI-19]{srfi.19} Returns an instance of <time> class with specified initial values. Equivalent to (make <time> :type type :second seconds :nanosecond nanoseconds).

- **time-type** time
- **time-second** time
- **time-nanosecond** time
- **set-time-type!** time type
- **set-time-second!** time second
- **set-time-nanosecond!** time nanosecond
  [SRFI-19]{srfi.19} Getter and setter of <time> object slots.
Chapter 11: Library modules - SRFIs

11.7.4 Date

`<date>`

{srfi.19} Represents a date.

- **nanosecond** [Instance Variable of `<date>`]
  Nanosecond portion of the date by an integer between 0 and 999,999,999, inclusive.

- **second** [Instance Variable of `<date>`]
  Second portion of the date by an integer between 0 and 60, inclusive. (60 for leap second).

- **minute** [Instance Variable of `<date>`]
  Minute portion of the date by an integer between 0 and 59, inclusive.

- **hour** [Instance Variable of `<date>`]
  Hour portion of the date by an integer between 0 and 23, inclusive.

- **day** [Instance Variable of `<date>`]
  Day portion of the date by an integer between 0 and 31, inclusive. The actual upper bound of the day is determined by the year and the month. (Note: 1 is for the first day; 0 is allowed by the specification, but I don’t see why).

- **month** [Instance Variable of `<date>`]
  Month portion of the date by an integer between 1 and 12, inclusive. 1 for January, 2 for February, and so on. (Note: this is different from POSIX’s `<sys-tm>` convention).

- **year** [Instance Variable of `<date>`]
  Year portion of the date.

- **zone-offset** [Instance Variable of `<date>`]
  The number of seconds east of GMT for this timezone, by an integer.
make-date  nanosecond second minute hour day month year zone-offset  [Function]
   [SRFI-19]{srfi.19} Makes a <date> object from the given values. Note: this procedure does not check if the values are in the valid range.

date?  obj  [Function]
   [SRFI-19]{srfi.19} Returns true iff obj is a <date> object.

date-nanosecond  date  [Function]
date-second  date  [Function]
date-minute  date  [Function]
date-hour  date  [Function]
date-day  date  [Function]
date-month  date  [Function]
date-year  date  [Function]
date-zone-offset  date  [Function]
   [SRFI-19]{srfi.19} Accessors.

date-year-day  date  [Function]
date-week-day  date  [Function]
date-week-number  date  day-of-week-starting-week  [Function]
   [SRFI-19]{srfi.19} Calculates the day number in the year (1 for January 1st), the day number in the week (0 for Sunday, 1 for Monday, ...), and the ordinal week of the year which holds this date, ignoring a first partial week, respectively.

   Day-of-week-starting-week is the integer corresponding to the day of the week which is to be considered the first day of the week (Sunday=0, Monday=1, etc.).

date->julian-day  date  [Function]
date->modified-julian-day  date  [Function]
date->time-monotonic  date  [Function]
date->time-tai  date  [Function]
date->time-utc  date  [Function]
   [SRFI-19]{srfi.19} Conversions from date to various date/time types.

julian-day->date  jd :optional tz-offset  [Function]
julian-day->time-monotonic  jd  [Function]
julian-day->time-tai  jd  [Function]
julian-day->time-utc  jd  [Function]
   [SRFI-19]{srfi.19} Conversions from julian-day to various date/time types.

modified-julian-day->date  jd :optional tz-offset  [Function]
modified-julian-day->time-monotonic  jd  [Function]
modified-julian-day->time-tai  jd  [Function]
modified-julian-day->time-utc  jd  [Function]
   [SRFI-19]{srfi.19} Conversions from modified julian-day to various date/time types.

time-monotonic->date  time :optional tz-offset  [Function]
time-monotonic->julian-day  time  [Function]
time-monotonic->modified-julian-day  time  [Function]
time-monotonic->time-tai  time  [Function]
time-monotonic->time-tai!  time  [Function]
time-monotonic->time-utc  time  [Function]
time-monotonic->time-utc!  time  [Function]
   [SRFI-19]{srfi.19} Conversions from time-monotonic to various date/time types.
Chapter 11: Library modules - SRFIs

11.7.5 Date reader and writer

date->string  date :optional format-string  [Function]

[SRFI-19][srfi.19] Converts a <date> object to a string, according to the format specified by format-string. If format-string is omitted, "~c" is assumed.

A format string is copied to output, except a sequence begins with ~ which is replaced with the following rules:

~A  Locale’s full weekday name (Sunday...Saturday).
~B  Locale’s full month name (January...December).
~C  Locale’s date and time (e.g., "Fri Jul 14 20:28:42-0400 2000").
~D  Day of month, zero padded (01...31).
~d  Day of month, zero padded (01...31).
~e  Day of month, blank padded ( 1...31).
~f  Seconds+fractional seconds, using locale’s decimal separator (e.g. 5.2).
~h  Same as ~b.
~H  Hour, zero padded, 24-hour clock (00...23).
~I  Hour, zero padded, 12-hour clock (01...12).
~j  Day of year, zero padded.
~k  Hour, blank padded, 24-hour clock (0...23).
~L  Hour, blank padded, 12-hour clock (1...12).
~m  Month, zero padded (01...12).
~M  Minute, zero padded (00...59).
~n  New line.
-N Nanosecond, zero padded.
-\ p Locale’s AM or PM.
-\ r Time, 12 hour clock, same as "~I:\M:\S \p".
-\ s Number of full seconds since "the epoch" (in UTC).
-\ \ S Second, zero padded (00...60).
-\ t Horizontal tab.
-\ T Time, 24 hour clock, same as "~H:\M:\S".
-\ U Week number of year with Sunday as first day of week (00...53).
-\ V ISO8601 Week number of year with Monday as first day of week. The week with the first Thursday is week 01. If there’s a partial week before that, it becomes week 52 or 53 of the preceding year (01...53).
-\ w Day of week (0...6).
-\ \ W Week number of year with Monday as first day of week (00...52).
-\ \ x Locale’s date representation, for example: *07/31/00*.
-\ \ X Locale’s time representation, for example: *06:51:44*.
-\ \ y Last two digits of year (00...99).
-\ \ Y Year.
-\ z Time zone in RFC-822 style.
-\ \ 1 ISO-8601 year-month-day format.
-\ \ 2 ISO-8601 hour-minute-second-timezone format.
-\ \ 3 ISO-8601 hour-minute-second format.
-\ \ 4 ISO-8601 year-month-day-hour-minute-second-timezone format.
-\ \ 5 ISO-8601 year-month-day-hour-minute-second format.

Note: currently Gauche doesn’t honor process’s locale setting, and it always formats the date as if the locale is "C". It may be changed in future, so you shouldn’t rely on, for example, ~a always formatted as "Sun"..."Sat".

There’s no portable way to ensure you’ll get "C" locale formats since there’s no standard way to set process’s locale yet. However, Gauche provides a way to ensure the locale to be "C", as an extension to srfi-19. Insert @ between ~ and the directive character, such as ~@a.

string->date string template-string [Function] [SRFI-19]{srfi.19}

11.8 srfi.27 - Sources of Random Bits

srfi.27 [Module]
This module provides SRFI-27 pseudo random generator interface, using Mersenne Twister algorithm (see Section 12.36 [Mersenne-Twister random number generator], page 893) as the backbone.
The random sources provided in this module is thread safe; multiple thread can use the same random source without breaking its internal state.
random-integer \( n \)  
[Function]  
[SRFI-27]{srfi.27} Returns a random exact integer between \([0, n-1]\), inclusive, using the default random source. To set a random seed for this procedure, use \texttt{random-source-randomize!} or \texttt{random-source-pseudo-randomize!} on \texttt{default-random-source}.

random-real  
[Function]  
[SRFI-27]{srfi.27} Returns a random real number between \((0, 1)\), exclusive, using the default random source. To set a random seed for this procedure, use \texttt{random-source-randomize!} or \texttt{random-source-pseudo-randomize!} on \texttt{default-random-source}.

default-random-source  
[Variable]  
[SRFI-27]{srfi.27} Keeps the default random source that is used by \texttt{random-integer} and \texttt{random-real}.

make-random-source  
[Function]  
[SRFI-27]{srfi.27} Creates and returns a new random source. In the current Gauche implementation, it is just a \texttt{<mersenne-twister>} object. It may be changed in the future implementation.

random-source? \ obj \)  
[Function]  
[SRFI-27]{srfi.27} Returns \#t if \ obj \ is a random source object.

random-source-state-ref \ s \)  
random-source-state-set! \ s \ state \)  
[Function]  
[Function]  
[SRFI-27]{srfi.27} Gets and sets the "snapshot" of the state of the random source \ s \. \ State \ is an opaque object whose content depends on the backbone generator.

random-source-randomize! \ s \)  
[Function]  
[SRFI-27]{srfi.27} Makes an effort to set the state of the random source \ s \ to a truly random state. The current implementation uses the current time and the process ID to set the random seed.

random-source-pseudo-randomize! \ s \ i \ j \)  
[Function]  
[SRFI-27]{srfi.27} Changes the state of the random source \ s \ into the initial state of the \((i, j)\)-th independent random source, where \( i \) and \( j \) are non-negative integers. This procedure can be used to reuse a random source \ s \ as large number of independent random source, indexed by two non-negative integers. Note that this procedure is entirely deterministic.

random-source-make-integers \ s \)  
[Function]  
[SRFI-27]{srfi.27} Returns a procedure, that takes one integer argument \( n \) and returns a random integer between 0 and \( n-1 \) inclusive for every invocation, from the random source \ s \.

random-source-make-reals \ s \ :optional unit \)  
[Function]  
[SRFI-27]{srfi.27} Returns a procedure, that takes no argument and returns a random real between 0 and 1 exclusive for every invocation, from the random source \ s \. If \ unit \ is given, the random real the returned procedure generates will be quantized by the given \ unit \, where \( 0 < \ unit < 1 \).

11.9 srfi.29 - Localization

srfi.29  
[Module]  
This module implements the message localization mechanism defined in SRFI-29.

In fact, this module consists of two submodules, srfi.29.bundle and srfi.29.format. The module srfi.29 extends both submodules. It is because srfi-29’s definition of the format
procedure is incompatible to Gauche’s native format (thus Common Lisp’s format) in the handling of ~@* directive.

So I splitted the module into two, srfi.29.format which contains srfi-29’s format, and srfi.29.bundle which contains the rest ("bundle" API). If a program wishes a complete compatibility of srfi-29, use srfi.29 module, which overrides Gauche’s native format. If a program just wants srfi-29’s "bundle" API, but wants to keep Gauche’s format, use srfi.29.bundle.

A localization feature is also provided by text.gettext module (see Section 12.69 [Localized messages], page 1000), which is a preferable way of message localization in Gauche. This module is provided mainly for porting code that uses srfi-29 features.

Bundle specifier

A bundle specifier is an arbitrary list of symbols, but typically it takes the form like:

(package language country details ...)

Where package specifies the software package, language and country specifies language and country code, and details gives other informations like encoding.

The values for the default bundle specifier can be obtained by the following parameters.

- current-language
- current-country
- current-locale-details

The current-language and current-country parameters keep the ISO 639-1 language code and ISO 3166-1 country code respectively, both as symbols. The current-locale-details keeps a list of auxiliary local informations, such as encodings.

These parameters are initialized if LANG environment variable is set in the form of lang_country.encoding format. For example, if the LANG setting is ja_JP.eucJP, those parameters are ja, jp, and (eucjp), respectively. If LANG is C or undefined, the default values are en, us, and (), respectively.

Bundle preparation

declare-bundle! bundle-specifier association-list

Put the association list of template key (symbol) and the locale-specific message (string) into the bundle database, with bundle-specifier as the key.

Gauche currently supports only in-memory bundle database. That is, you have to call declare-bundle! within the application in order to lookup the localized messages.

save-bundle! bundle-specifier
load-bundle! bundle-specifier

Since Gauche doesn’t support persistent bundle database yet, these procedures does nothing and returns #f. (It is still conforming behavior of srfi-29).

Retrieving localized message

localized-template package-name message-template-name

Retrieves localized message, associated with a symbol message-template-name in the package package-name.
Extended format procedure

format format-string args                      [Function]
[SRFI-29]{srfi.29} SRFI-29 extends SRFI-28’s format procedure spec (which supports "a, "s, "h and "r directives), in order to support argument repositioning.

A directive "N@*", where N is an integer or can be omitted, causes the next directive to retrieve a value from N-th optional argument. The referenced value isn’t consumed, and won’t affect the processing of subsequent directives.

Although SRFI-28 spec is compatible to Gauche’s native format (see Section 6.21.8 [Output], page 275), this SRFI-29 extension isn’t. Specifically, the "N@*" directive of Gauche’s format changes the argument pointer to points N-th optional argument, thus it affects all the subsequent arguments.

Because of this incompatibility, this function is defined in a separate module, srfi.29.format. If you use srfi.29, which extends srfi.29.bundle and srfi.29.format, the format procedure will be overridden by srfi-29’s format in your module. If you want to keep Gauche’s native format, use srfi.29.bundle only.

11.10 srfi.37 - args-fold: a program argument processor

srfi.37                                              [Module]
This module implements args-fold, yet another procedure to process command-line arguments, defined in SRFI-37.

Unlike gauche.parseopt (see Section 9.25 [Parsing command-line options], page 480), args-fold provides functional interface, i.e. the user’s states are explicitly passed via parser’s argument and return values, and also follows POSIX and GNU getopt guidelines, including long options.

args-fold args options unrecognized-proc operand-proc :rest seeds                      [Function]
[SRFI-37]{srfi.37} Processes program options args from left to right, according to given option specification options, and two procedures unrecognized-proc and operand-proc.

Options is a list of option objects, explained below. Each option object keeps the name(s) of the option, a flag to specify whether the option takes an argument or not, and a procedure to process that option (we’ll call it option procedure).

Args-fold recognizes both single-character options (short options) and long options. A short option must begin with single hyphen (e.g. -a), while long option must begin with double hyphens (e.g. --help). Short options can be concatenated, e.g. -a bc or -a -b -c. Both a short option and a long option can take required or optional arguments. Required short-option argument can appear with or without space after the option, e.g. -afoo or -a foo. Long-option argument can appear after character '=' or space, e.g. --long=foo or --long foo.

When args-fold encounters a command-line argument that cannot be an option argument, and doesn’t begin with hyphen, the argument is treated as an operand. Args-fold allows operands and options to be interleaved. However, if args-fold encounters '--', the rest of arguments are treated as operands, regardless of beginning with hyphen or not.

When the given option matches one of option object in options, the option procedure is called as follows:

(option-proc option name arg seed ...)

where option is the matched option object, name is the string actually used to specify the option, arg is the option argument (or #f if there’s none), and seed ... is the user’s state information. Option-proc must return as many arguments as seeds.
When \texttt{args-fold} encounters an option that doesn’t match any of the option objects, it creates a new option object for the option and calls \texttt{unrecognized-proc} with the same arguments as \texttt{option-proc}.

When \texttt{args-fold} finds an operand, \texttt{operand-proc} is called as follows:

\[
(\text{operand-proc} \ \text{operand} \ \text{seed} \ \ldots)
\]

\textit{Operand-proc} must return as many arguments as \textit{seed}s.

The caller’s state should be explicitly passed around seed arguments and return values. The initial seed values are \textit{seeds} given to \texttt{args-fold}. The values returned from option procedure, \texttt{unrecognized-proc} and \texttt{operand-proc} are used as the seed arguments of next invocation of those procedures. The values returned from the last call to the procedures are returned from \texttt{args-fold}.

\begin{verbatim}
\textbf{option} names require-arg? optional-arg? processor
\{SRFI-37\}

\texttt{Option names require-arg? optional-arg? processor} creates an option object with the passed properties.

\textit{Names} is a list of characters and/or strings. A character is used for a short option, and a string is used for a long option.

Two flags, \textit{require-arg?} and \textit{optional-arg?} indicates whether the option should take an option argument, or may take an option argument.

\textit{Processor} is the option processor procedure.

Note that, if an option argument is passed using \texttt{"-"} character, it is passed to the option procedure even if the option has \texttt{#f} in both \textit{require-arg?} and \textit{optional-arg?}. It is up to the option procedure to deal with the argument.

It should also be noted that the optional option argument for a short option is only recognized if it is given without whitespace after the short option. That is, if a short option \texttt{"d"} is marked to take optional option argument, then \texttt{"-dfoo"} is interpreted as \texttt{"-d"} with argument \texttt{"foo"}, but \texttt{"-d foo"} is interpreted as \texttt{"-d"} without argument and an operand \texttt{foo}. If \texttt{"d"} is marked to take required option argument, however, both are interpreted as \texttt{"-d"} with argument \texttt{"foo"}.

\end{verbatim}

\begin{verbatim}
\textbf{option?} obj
\{SRFI-37\}

\texttt{Option?} \textit{obj} returns \texttt{#t} if \textit{obj} is an option object, \texttt{#f} otherwise.

\end{verbatim}

\begin{verbatim}
\textbf{option-names} option
\textbf{option-required-arg?} option
\textbf{option-optional-arg?} option
\textbf{option-processor}
\{SRFI-37\}

\texttt{Option-names} \textit{option} \texttt{option-required-arg?} \textit{option} \texttt{option-optional-arg?} \textit{option} \texttt{option-processor} returns the properties of an option object \textit{option}.

\end{verbatim}

A simple example:

\begin{verbatim}
(\use srfi.37)

(define options (list
  (option '(#\d "debug") #f #t
    (lambda (option name arg debug batch paths files)
      (values (or arg "2") batch paths files)))
  (option '(#\b "batch") #f #f
    (lambda (option name arg debug batch paths files)
      (values debug #t paths files)))
  (option '(#\I "include") #t #f
    (lambda (option name arg debug batch paths files)
      (values debug batch (cons arg paths) files)))))
\end{verbatim}
(define (main args)
  (receive (debug-level batch-mode include-paths files)
    (args-fold (cdr args)
      options
        (lambda (option name arg . seeds) ; unrecognized
          (error "Unrecognized option:" name))
        (lambda (operand debug batch paths files) ; operand
          (values debug batch paths (cons operand files)))
        0 ; default value of debug level
        #f ; default value of batch mode
        '() ; initial value of include paths
        '() ; initial value of files
      )
      (print "debug level = " debug-level)
      (print "batch mode = " batch-mode)
      (print "include paths = " (reverse include-paths))
      (print "files = " (reverse files))
    0))

11.11 srfi.42 - Eager comprehensions

srfi.42

This module provides a generic comprehension mechanism, which some other languages (e.g. Haskell and Python) offer as a built-in mechanism. It provides a rich set of operators so it can be used not only as a list generator but as a generic loop construct (actually, some may say it is as powerful/evil as Common Lisp’s loop macro).

It runs eagerly as the name suggests, that is, if it generates a list, it creates the entire list when evaluated, instead of generate the elements on demand. Thus it can’t represent an infinite sequence, which Haskell’s comprehension naturally does. Gauche offers a few alternatives to deal with lazy, possibly infinite, sequences: See Section 6.18.2 [Lazy sequences], page 241, Section 9.12 [Generators], page 436, and Section 12.90 [Stream library], page 1033.

Eager comprehension examples

Let’s begin with some examples.

Generate a list of squares for the first five integers:

(list-ec (: i 5) (* i i)) ⇒ (0 1 4 9 16)

list-ec is a comprehension macro that generates a list. The first form (: i 5) is called a qualifier, which specifies a set of values to repeat over (here it is each integer from 0 below 5). The last form (* i i) is called a body, which is an ordinary Scheme expression evaluated for each values generated by the qualifier.

A comprehension can have more than one qualifiers. Next example generate set of pair of numbers (x y), where x is between 2 (inclusive) and 5 (exclusive), and y is between 1 (inclusive) and x (exclusive).

(list-ec (: x 2 5) (: y 1 x) (list x y))
⇒ ((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))

The qualifiers works as nested; that is, (: x 2 5) specifies to repeat the rest of the clauses—(: y 1 x) and (list x y).

The above two examples can be written in Haskell as the followings:

[ i*i | i <- [0..4] ]
[ (x,y) | x <- [2..4], y <- [1..x-1] ]
Note the differences: (1) In Haskell, the body expression to yield the elements comes first, followed by qualifiers (selectors). In SRFI-42, the body expression comes last. (2) In SRFI-42, range operator’s lower bound is inclusive but its upper bound is exclusive.

List a set of numbers \((a \ b \ c \ d)\), where \(a^3+b^3 = c^3+d^3\):

\[
\text{(define (taxi-number n)}
\text{(list-ec (: a 1 n)}
\text{  (: b (+ a 1) n)}
\text{  (: c (+ a 1) b)}
\text{  (: d (+ c 1) b)}
\text{  (if (= (+ (expt a 3) (expt b 3)}
\text{        (+ (expt c 3) (expt d 3))))
\text{  (list a b c d))})
\]

If you want to change values of more than one variable simultaneously, instead of nesting, you can bundle the qualifiers like this:

\[
\text{(list-ec (:parallel (: x '(a b c d)) (: y '}(1 2 3 4)))
\text{  (list x y))}
\]

⇒ \(((a 1) (b 2) (c 3) (d 4))\)

You can generate not only a list, but other sequences:

\[
\text{(vector-ec (: i 5) i)} \Rightarrow \text{#(0 1 2 3 4)}
\]
\[
\text{(string-ec (: i 5) (integer->char (+ i 65)))} \Rightarrow "ABCDE"
\]

Or apply folding operations:

\[
\text{(sum-ec (: i 1 100) i)} \Rightarrow 4950 ;; \text{sum of integers from 1 below 100.}
\]
\[
\text{(product-ec (: i 1 10) i)} \Rightarrow 362880 ;; \text{... and product of them.}
\]

**Comprehension macros**

Each comprehension takes the following form.

\[
\text{(comprehension-macro qualifier ... body)}
\]

It evaluates \text{body} repeatedly as specified by \text{qualifier} ... . Depending on the type of comprehension, the results of \text{body} may be either collected to create an aggregate (list, vector, string, ...), folded by some operator (sum, product, min, max, ...), or simply discarded.

Each \text{qualifier} specifies how to repeat the following \text{qualifiers} and \text{body}. A \text{qualifier} can be a generational qualifier that yields a set of values to loop over, or a control qualifier that specify a condition to exclude some values. See the Qualifiers heading below.

A few comprehensions takes extra values before \text{qualifiers} or after \text{body}.

\text{do-ec} \text{qualifier ... body} [Macro]

\[\text{SRFI-42}\{\text{srfi.42}\}\] Repeats \text{body}. The results of \text{body} is discarded. This is for side-effecting operations.

\text{list-ec} \text{qualifier ... body} [Macro]

\[\text{SRFI-42}\{\text{srfi.42}\}\] Repeats \text{body} and collects the results into a list.

\text{append-ec} \text{qualifier ... body} [Macro]

\[\text{SRFI-42}\{\text{srfi.42}\}\] Repeats \text{body}, which must yield a list. Returns a list which is the concatenation of all lists returned by \text{body}.

\text{string-ec} \text{qualifier ... body} [Macro]

\text{string-append-ec} \text{qualifier ... body} [Macro]

\[\text{SRFI-42}\{\text{srfi.42}\}\] Repeats \text{body}, which must yield a character (in \text{string-ec}) or a string (in \text{string-append-ec}). Returns a string that consists of the results of \text{body}. 
Chapter 11: Library modules - SRFIs

**vector-ec** qualifier ... body

[SRFI-42]{{srfi.42}} Repeats body and collects the results into a vector.

**vector-of-length-ec** k qualifier ... body

[SRFI-42]{{srfi.42}} This is like vector-ec, except that the length of the result vector is known to be k. It can be more efficient than vector-ec. Unless the comprehension repeats exactly k times, an error is signaled.

**sum-ec** qualifier ... body

**product-ec** qualifier ... body

[SRFI-42]{{srfi.42}} body must yield a numeric value. Returns sum of and product of the results, respectively.

**min-ec** qualifier ... body

**max-ec** qualifier ... body

[SRFI-42]{{srfi.42}} body must yield a real number. Returns maximum and minimum value of the results, respectively. body must be evaluated at least once, or an error is signaled.

**any?-ec** qualifier ... test

**every?-ec** qualifier ... test

[SRFI-42]{{srfi.42}} Evaluates test for each iteration, and returns #t as soon as it yields non-#f (for any?-ec), or returns #f as soon as it yields #f (for every?-ec). Unlink the comprehensions introduced above, these stop evaluating test as soon as the condition meets. If the qualifiers makes no iteration, #f and #t are returned, respectively.

**first-ec** default qualifier ... body

**last-ec** default qualifier ... body

[SRFI-42]{{srfi.42}} First initializes the result by the value of the expression default, then start iteration, and returns the value of the first and last evaluation of body, respectively. In fact, first-ec only evaluates body at most once.

These forms are most useful when used with control qualifiers. For example, the following first-ec returns the first set of distinct integers (x, y, z), where x*x+y*y+z*z becomes a square of another integer w.

```
(first-ec #f (:integers w) (: z 1 w) (: y 1 z) (: x 1 y)
  (if (= (* w w) (+ (* x x) (* y y) (* z z))))
  (list x y z w))
```

Note that the first qualifier, (:integers w), generates infinite number of integers; if you use list-ec instead of first-ec it won’t stop.

**fold-ec** seed qualifier ... expr proc

**fold3-ec** seed qualifier ... expr init proc

[SRFI-42]{{srfi.42}} Reduces the values produced by expr.

Suppose expr produces a sequence of values x0, x1, ..., xN. Fold-ec calculates the following value:

```
(prox0seedx00seed)....)
```

It’s similar to fold, except that proc is evaluated within the scope of qualifier ... so you can refer to the variables introduced by them. On the other hand, seed is outside of the scope of qualifiers.

Fold-ec3 is almost the same but the initial value calculation. In fold-ec3, seed is only used when qualifiers makes no iteration. Otherwise it calculates the following value:

```
(proxN...(proc x1 (init x0))...))
```
Qualifiers

Generational qualifiers

This type of qualifiers generates (possibly infinite) values over which the rest of clauses iterate.

In the following descriptions, vars refers to either a single identifier, or a series of identifier and a form (index identifier2). The single identifier in the former case and the first identifier in the latter case name the variable to which each generated value is bound. The identifier2 in the latter case names a variable to which a series of integers, increasing with each generated element, is bound. See the following example:

\[
\begin{align*}
& \text{(list-ec (\: x\ ' (a b c)) x)} \\
& \Rightarrow (a \ b \ c) \\
& \text{(list-ec (\: x (index y) ' (a b c)) (cons x y))} \\
& \Rightarrow ((a . 0) (b . 1) (c . 2))
\end{align*}
\]

\text{:\ vars arg1 args . . .}

A generic dispatcher of generational qualifiers. An appropriate generational qualifier is selected based on the types of arg1 args . . . .

\text{:\list vars arg1 args . . .} \quad \text{[EC Qualifier]}

\text{:\vector vars arg1 args . . .} \quad \text{[EC Qualifier]}

\text{:\uvector vars arg1 args . . .} \quad \text{[EC Qualifier]}

\text{:\string vars arg1 args . . .} \quad \text{[EC Qualifier]}

Arg1 args . . . should be all lists, vectors, uniform vectors or strings, respectively. Repeats the subsequent clauses while binding each element from those args bound to vars. (The :uvector qualifier is Gauche’s extension.)

\[
\text{(list-ec (\:string c "ab" "cd") c)} \Rightarrow (#\a #\b #\c #\d)
\]

If the arguments given to the generic qualifier : are all lists, vectors, uniform vectors or strings, then these qualifiers are used.

\text{:\integers vars} \quad \text{[EC Qualifier]}

Generates infinite series of increasing integers, starting from 0.

\text{:\range vars stop} \quad \text{[EC Qualifier]}

\text{:\range vars start stop} \quad \text{[EC Qualifier]}

\text{:\range vars start stop step} \quad \text{[EC Qualifier]}

\text{:\range vars range} \quad \text{[EC Qualifier]}

The first three forms generates a series of exact integers, starting from start (defaults to 0) and stops below stop, stepping by step (defaults to 1). Giving a negative integer to step makes a decreasing series.

\[
\begin{align*}
& \text{(list-ec (\:range v 5) v)} \Rightarrow (0 \ 1 \ 2 \ 3 \ 4) \\
& \text{(list-ec (\:range v 3 8) v)} \Rightarrow (3 \ 4 \ 5 \ 6 \ 7) \\
& \text{(list-ec (\:range v 1 8 2) v)} \Rightarrow (1 \ 3 \ 5 \ 7) \\
& \text{(list-ec (\:range v 8 1 -2) v)} \Rightarrow (8 \ 6 \ 4 \ 2)
\end{align*}
\]

If one, two or three exact integer(s) is/are given to the generic qualifier :, this qualifier is used.

If a range object (see Section 12.21 [Range], page 845) is given to this qualifier, as in the fourth form, this generates each element in the range sequentially.

\text{:\real-range vars stop} \quad \text{[EC Qualifier]}

\text{:\real-range vars start stop} \quad \text{[EC Qualifier]}
::real-range vars start stop step  
Generates a series of real numbers, starting from start (defaults to 0) and stops below stop, stepping by step (defaults to 1). If all the arguments are exact numbers, the result consists of exact numbers; if any of the arguments are inexact, the result consists of inexact numbers.

**(list-ec (:real-range v 5.0) v)**
⇒ (0.0 1.0 2.0 3.0 4.0)

**(list-ec (:real-range v 1 4 1/3) v)**
⇒ (1 4/3 5/3 2 7/3 8/3 3 10/3 11/3)

**(list-ec (:real-range v 1 5.0 1/2) v)**
⇒ (1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5)

If one, two or three real numbers is/are given to the generic qualifier :; and any one of them isn’t an exact integer, then this qualifier is used.

::char-range vars min max  
Generates a series of characters, starting from min and ending at max (inclusive). The characters are enumerated in the order defined by char<=? (see Section 6.9 [Characters], page 160).

**(list-ec (:char-range v #\a #\e) v)**
⇒ (#\a #\b #\c #\d #\e)

If two characters are given to the generic qualifier :; this qualifier is used.

::port vars port  
::port vars port read-proc  
Generates a series of values read from an input port port, by the procedure read-proc (defaults to read). The series terminates when EOF is read.

**(call-with-input-string "a \"b\" :c**

(p (list-ec (:port v p) v)))
⇒ (a "b" :c)

If one or two arguments are given to the generic qualifier :; and the first one is an input port, then this qualifier is used.

::generator vars gen  
This is Gauche’s extension and not defined in SRFI-42. gen must be a procedure with zero arguments. This qualifier repeats until gen returns EOF.

Gauche has a set of utilities to create and operate on such procedures; see Section 9.12 [Generators], page 436.

**(use gauche.generator)**

**(list-ec (:generator v (grange 1 8)) v)**
⇒ (1 2 3 4 5 6 7)

If one argument is given to the generic qualifier :; and it is applicable without arguments, then this qualifier is used.

::collection vars coll  
This is Gauche’s extension and not defined in SRFI-42. coll must be an instance of <collection> or its subclass. (see Section 9.6 [Collection framework], page 404). This qualifier repeats over the elements in the collection.

If srfi-42 has a specialized qualifier, it is faster to use it (e.g. a vector is also a collection, but using :vector is faster than :collection).

If one argument is given to the generic qualifier :; and it is a collection other than the specific types supported directly in SRFI-42, this qualifier is used.
:parallel generator . . .
This is used to run through multiple generators in parallel. It terminates when any one of
 generator is exhausted.

\[
\text{(list-ec (:parallel (: x ' (a b c))
\quad (: y "defg"))}
\quad (cons x y))
\Rightarrow ((a . #\d) (b . #\e) (c . #\f))
\]

;; Compare with this:
\[
\text{(list-ec (: x ' (a b c))}
\quad (: y "defg")
\quad (cons x y))
\Rightarrow ((a . #\d) (a . #\e) (a . #\f) (a . #\g)
\quad (b . #\d) (b . #\e) (b . #\f) (b . #\g)
\quad (c . #\d) (c . #\e) (c . #\f) (c . #\g))
\]

:let vars expr
Evaluate expr, and bind the result to vars and execute the following clauses once. If vars has
an index var, it is bound to 0. It is effectively the same as (:list vars (list expr)).

:while g-qualifier expr
:until generator expr
Generates values from g-qualifier while/until expr evaluates true. Here, g-qualifier is one of
generative EC qualifiers. The variable bound in g-qualifier is visible from expr.

\[
\text{(use math.prime)}
\quad \text{(sum-ec (:while (: p (index k) *primes*) (<= k 100)) p)}
\]

:dispatched vars dispatch arg1 args . . .
:do (lb . . .) ne1? (ls . . .)
:do (let (ob . . .) oc . . .) (lb . . .) ne1? (let (ib . . .) ic . . .) ne2? (ls . . .)

Control qualifiers

if test
Evaluates test, and if it yields false, stops that iteration and start the next iteration.
The following examples returns a list of pythagorian triplets (a b c), which satisfies a^2 +
b^2 = c^2, less than 100.

\[
\text{(list-ec (: c 1 100) (: a 1 c) (: b a c)}
\quad \text{(if (= (square c) (+ (square a) (square b)))}}
\quad \text{list a b c))}
\Rightarrow ((3 4 5) (6 8 10) (5 12 13) (9 12 15) (8 15 17) . . .)
\]

not test
and test . . .
or test . . .
A shorthand of (if (not test)), (if (and test . . .)), and (if (or test . . .)).

begin command . . . expr
During iteration, evaluates command . . . for side effects before evaluates expr. The result of
commands are discarded.
nested qualifier . . . [EC Qualifier]

Splices qualifier . . . For example, \((\text{list-ec} (: \text{a} 2) (\text{nested} (: \text{b} 2) (: \text{c} 2)) (: \text{d} 2) (\text{list} \text{a} \text{b} \text{c} \text{d}))\) is equivalent to \((\text{list-ec} (: \text{a} 2) (\text{nested} (: \text{b} 2) (: \text{c} 2) (: \text{d} 2) (\text{list} \text{a} \text{b} \text{c} \text{d}))\).

11.12 srfi.43 - Vector library (legacy)

srfi.43 [Module]

This module is effectively superseded by R7RS and SRFI-133. There are a few procedures that are not compatible with R7RS and SRFI-133, and this module remains to support legacy code that depends on them.

See Section 6.13.1 [Vectors], page 195, and see Section 10.3.2 [R7RS vectors], page 597, for the “modern” versions of vector library. New code should use them.

The following procedures in SRFI-43 are built-in. See Section 6.13.1 [Vectors], page 195, for the description.

- make-vector  vector  vector?  vector-ref
- vector-set!  vector-length  vector-fill!  vector-copy
- vector-copy!  vector-append  vector->list  list->vector
- reverse-list->vector

The following procedures in SRFI-43 are supported by SRFI-133. See Section 10.3.2 [R7RS vectors], page 597, for the description.

- vector-unfold  vector-unfold-right  vector-reverse-copy
- vector-reverse-copy!  vector-concatenate  vector-empty?
- vector=  vector-index  vector-index-right
- vector-skip  vector-skip-right  vector-binary-search
- vector-any  vector-every  vector-swap!
- reverse-vector->list

We explain the procedures that are not listed above.

\texttt{vector-fold} \texttt{kons} \texttt{knil} \texttt{vec1} \texttt{vec2} . . . \[Function\]
\texttt{vector-fold-right} \texttt{kons} \texttt{knil} \texttt{vec1} \texttt{vec2} . . . \[Function\]

[SRFI-43]srfi.43 Like \texttt{vector-fold} and \texttt{vector-fold-right} in SRFI-133, but \texttt{kons} takes an extra argument, the current index, as its first argument. So \texttt{kons} must accept \(n+2\) arguments, where \(n\) is the number of given vectors. It is called as \((\texttt{kons} \\texttt{<index>} \texttt{<cumulated-value>} \texttt{<elt1>} \texttt{<elt2>} \ldots)\).

Gauche has \texttt{fold-with-index} (see Section 9.31.3 [Mapping over sequences], page 511) that can be used to fold vectors with index, but the argument order of \texttt{kons} is slightly different: It passes the index, each element from argument vectors, then cumulated values.

\begin{verbatim}
(use srfi.43)
(vector-fold list '() '#(a b c) '#(d e f))
⇒ (2 (1 (0 () a d) b e) c f)

(use gauche.sequence)
(fold-with-index list '() '#(a b c) '#(d e f))
⇒ (2 c f (1 b e (0 a d ()())
\end{verbatim}

\texttt{vector-map} \texttt{f} \texttt{vec1} \texttt{vec2} . . . \[Function\]
\texttt{vector-map!} \texttt{f} \texttt{vec1} \texttt{vec2} . . . \[Function\]
\texttt{vector-for-each} \texttt{f} \texttt{vec1} \texttt{vec2} . . . \[Function\]
vector-count \( f \) vec1 vec2 ... [Function]

[SRFI-43]{srfi.43} Like vector-map and vector-for-each of R7RS, and vector-map! and vector-count in srfi-133, except \( f \) takes an extra argument, the current index, as its first argument.

Gauche provides vector-map-with-index, vector-map-with-index! and vector-for-each-with-index which are the same as SRFI-43’s vector-map, vector-map! and vector-for-each, respectively. See Section 6.13.1 [Vectors], page 195.

```scheme
(vector-map list ’(a b c))
⇒ #((0 a) (1 b) (2 c))
(vector-map list ’(a b c) ’(d e f g))
⇒ #((0 a d) (1 b e) (2 c f))
(vector-count = ’(0 2 2 4 4))
⇒ 3
```

(Note: The vector-count example calls = with two arguments, the current index and the element, for each element of the input vector. So it counts the number of occasions when the element is equal to the index.)

The generic map and for-each in gauche.collection can be used on vectors, but the mapped procedure is called without index, and the result is returned as a list. (vector-map \( f \) vec1 vec2 ...) is operationally equivalent to (map-to-with-index <vector> \( f \) vec1 vec2 ...). See Section 9.6 [Collection framework], page 404, and Section 9.31 [Sequence framework], page 510.

11.13 srfi.55 - Requiring extensions

srfi.55 [Module]

This module defines require-extension macro, a yet another way to write portable scripts. See Section 4.12 [Feature conditional], page 75, and Section 11.5 [Feature-based program configuration language], page 691, for other means of ensuring specific features.

This module is autoloaded when you use require-extension, so you don’t need explicitly say (use srfi.55); for portable scripts, you shouldn’t.

require-extension clause ...

[SRFI-55]{srfi.55} Make extension(s) specified by clauses available in the rest of the program.

A clause takes the following form:

```
(extension-id extension-arg ...)
```

Currently, only srfi is supported as extension-id, and its arguments are SRFI numbers.

For example, the following form:

```
(require-extension (srfi 1 13 14))
```

Roughly corresponds to Gauche’s use forms:

```
(use srfi.1)
(use srfi.13)
(use srfi.14)
```

11.14 srfi.60 - Integers as bits

srfi.60 [Module]

This srfi provides bit operations on integers, regarding them as 2’s complement representation. It is compatible to SLIB’s logical module.
The newer SRFI-151 (see Section 10.3.22 [R7RS bitwise operations], page 664) provides the same functionality and more, with more consistent naming. We recommend new code to use SRFI-151, while we keep SRFI-60 for the backward compatibility.

The following procedures are Gauche built-in. See Section 6.3.6 [Basic bitwise operations], page 136, for the description.

- \texttt{lognot}
- \texttt{logand}
- \texttt{logior}
- \texttt{logxor}
- \texttt{logtest}
- \texttt{logcount}
- \texttt{integer-length}
- \texttt{logbit?}
- \texttt{copy-bit}
- \texttt{bit-field}
- \texttt{ash}

The following procedures are defined in SRFI-151. See Section 10.3.22 [R7RS bitwise operations], page 664, for the description.

- \texttt{bitwise-not}
- \texttt{bitwise-and}
- \texttt{bitwise-ior}
- \texttt{bitwise-xor}
- \texttt{arithmetic-shift}
- \texttt{bit-count}
- \texttt{bitwise-if}
- \texttt{bit-set?}
- \texttt{copy-bit}
- \texttt{first-set-bit}

We describe procedures that are unique in SRFI-60 below.

\textbf{bitwise-merge} \quad \texttt{mask n0 n1} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{Same as \texttt{bitwise-if} (see Section 10.3.22 [R7RS bitwise operations], page 664).}

\textbf{any-bits-set?} \quad \texttt{mask n} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{Same as builtin \texttt{logtest} (see Section 6.3.6 [Basic bitwise operations], page 136). It is also called \texttt{any-bit-set?} in SRFI-151 (see Section 10.3.22 [R7RS bitwise operations], page 664).}

\textbf{log2-binary-factors} \quad \texttt{n} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{It is also called as \texttt{first-set-bit} in this srfi, which is also in SRFI-151 (see Section 10.3.22 [R7RS bitwise operations], page 664). This is equivalent to Gauche’s built-in \texttt{twos-exponent-factor} (see Section 6.3.6 [Basic bitwise operations], page 136).}

\textbf{rotate-bit-field} \quad \texttt{n count start end} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{This is equivalent to \texttt{bit-field-rotate} in SRFI-151 (see Section 10.3.22 [R7RS bitwise operations], page 664).}

\textbf{reverse-bit-field} \quad \texttt{n start end} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{This is equivalent to \texttt{bit-field-reverse} in SRFI-151 (see Section 10.3.22 [R7RS bitwise operations], page 664).}

\textbf{integer->list} \quad \texttt{n :optional len} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{Breaks \texttt{n} to each bits, representing 1 as \#t and 0 as \#f, LSB last, and returns a list of them. If a nonnegative integer \texttt{len} is given, it specifies the length of the result. If it is omitted, \texttt{(integer-length \texttt{n})} is used.}

\begin{align*}
(\text{integer->list 10}) & \Rightarrow (\#t \#f \#t \#f) \\
(\text{integer->list 10 6}) & \Rightarrow (\#f \#f \#f \#f \#f \#f)
\end{align*}

SRFI-151 has similar procedure \texttt{bits->list}, with a reversed bit order (LSB first) (see Section 10.3.22 [R7RS bitwise operations], page 664).

\textbf{list->integer} \quad \texttt{lis} \quad \texttt{[Function]} \quad \texttt{[SRFI-60]{srfi.60}} \quad \texttt{Takes a list of boolean values, replaces the true value for 1 and the false value for 0, and compose an integer regarding each value as a binary digit. If \texttt{n} is nonnegative integer, \texttt{(eqv? (list->integer (integer->list \texttt{n})) \texttt{n})} is true.}

\begin{align*}
(\text{list->integer '(\#f \#t \#f \#t \#f)}) & \Rightarrow 10
\end{align*}

SRFI-151 has similar procedure \texttt{list->bits}, with a reversed bit order (LSB first) (see Section 10.3.22 [R7RS bitwise operations], page 664).
booleans->integer  bool ...  [Function]
[SRFI-60]{srfi.60} ≡ (list->integer (list bool ...))

SRFI-151 has similar procedure bits, with a reversed bit order (LSB first) (see Section 10.3.22 [R7RS bitwise operations], page 664).

11.15 srfi.64 - A Scheme API for test suites

srfi.64  [Module]

This module defines API to write a portable test suite. In Gauche, it is adapted to work with gauche.test native test framework (see Section 9.34 [Unit testing], page 521).

If SRFI-64 tests are run with the default runner during gauche.test is active, the tests becomes a part of the whole gauche.test suite.

The recommended way is to write a test suite in pure srfi-64, then include it from the gauche test script:

```
(use gauche.test)
(test-start "the tests")
;; portable test
(include "test-suite-in-srfi-64")

;; gauche-specific test, if needed
...
```

(test-end)

If test-suite-in-srfi-64.scm is run by itself, it uses SRFI-64’s default reporting system.
If it is run within gauche.test script, the results are reported via gauche.test, consolidated with other Gauche test results.

11.15.1 Test API

11.15.2 Test runner

11.16 srfi.66 - Octet vectors

srfi.66  [Module]

This module defines procedures to deal with u8vectors; they are almost a subset of srfi.160 and gauche.uvector (see Section 6.13.2 [Uniform vectors], page 198, except one procedure, u8vector-copy!, which has different argument orders (unfortunate historical artifacts).

There’s no reason to use this srfi except porting code that relies on srfi.66.

The following procedures are the same as gauche.uvector:

```
u8vector?  make-u8vector  u8vector
u8vector->list  list->u8vector
u8vector-length  u8vector-ref  u8vector-set!
u8vector=?  u8vector-compare  u8vector-copy
```

u8vector-copy!  src sstart target tstart n  [Function]
[SRFI-66]{srfi.66} Copy the content of an u8vector src, starting from sstart for n octets, into an u8vector target beginning from tstart. The target u8vector must be mutable.

Note that gauche.uvector has also u8vector-copy!, but its argument order is as follows, where send is (+ sstart n):

```
(u8vector-copy! target tstart src sstart send)
```
Gauche’s argument order is consistent with `vector-copy!` of R7RS, SRFI-43 and SRFI-133. We recommend to use `srfi.66` only for porting third-party libraries to avoid confusion.

### 11.17 srfi.69 - Basic hash tables

**srfi.69**

This module has been superseded by R7RS `scheme.hash-table` (see Section 10.3.7 [R7RS hash tables], page 618). New code should use it instead.

This is a thin adaptor on Gauche’s built-in hashtables (see Section 6.14.1 [Hashtables], page 206). This is provided for the compatibility to the portable libraries; the hashtable object created by this module’s `make-hash-table` is the same as the one created by Gauche’s built-in, and you can pass the table to both APIs.

Here’s a summary of difference between SRFI-69 and Gauche’s built-in hash table API:

- The constructor `make-hash-table`, as well as `alist->hash-table`, takes equality predicate and hash function, instead of a single comparator argument as Gauche does.
- The hash function passed to SRFI-69’s `make-hash-table` takes two arguments, an object to calculate a hash value, and a positive integer that limits the range of the hash value.
- SRFI-69’s primary hash table accessor is `hash-table-ref`, which takes a thunk to be called when the table doesn’t have an entry for the given key, while Gauche’s `hash-table-get` takes a fallback value for that. SRFI-69 also has `hash-table-ref/default`, which takes a fallback value like Gauche’s `hash-table-get`, but the default value can’t be omitted.
- The basic iterator of SRFI-69 is called `hash-table-walk`, which is Gauche’s `hash-table-for-each`. The srfi name is chosen to avoid conflict with existing Scheme implementations.

The following procedures are the same as Gauche’s built-in ones. See Section 6.14.1 [Hashtables], page 206, for the details.

```scheme
hash-table? hash-table-delete! hash-table-exists?
hash-table-keys hash-table-values hash-table-fold
hash-table->alist hash-table-copy
```

**make-hash-table**

This is the same name as Gauche’s built-in procedure, but the arguments are different.

The `eq-pred` argument is an equality predicate; it takes two arguments and returns `#t` if two are the same, and `#f` if not. When omitted, `equal?` is used.

The `hash-proc` argument is a hash function. It takes two arguments: an object to hash, and a positive integer to limit the range of the hash value. (Note that Gauche’s native hash functions takes only one argument.) When omitted, Gauche tries to choose appropriate hash function if `eq-pred` is known one (e.g., `eq?`, `eqv?`, `equal?`, `string=?` or `string-ci=?`). Otherwise we use `scheme.hash-table's hash` procedure (see Section 10.3.7 [R7RS hash tables], page 618). But there’s no guarantee that the generic `hash` works appropriately; you should give suitable `hash-proc` if you pass custom `eq-pred`.

The returned hash table is an instance of Gauche’s native hash table. You can pass it to Gauche’s built-in procedures.

SRFI-69 allows implementation-specific arguments `args` to be passed to `make-hash-table`. At this moment, Gauche ignores them.
alist->hash-table  alist :optional eq-pred hash-fn :rest args  [Function]

[SRFI-69]{srfi.69} Like Gauche’s builtin alist->hash-table, but takes eq-pred and hash-fn separately, instead of a single comparator.

The alist argument is a list of pairs. The car of each pair is used for a key, and the cdr for its value.

See make-hash-table above for the description of eq-pred, hash-fn and args.

hash-table-equivalence-function  ht  [Function]
hash-table-hash-function  ht  [Function]

[SRFI-69]{srfi.69} Returns equivalence function and hash function of the hashtable ht.

Note that SRFI-69’s hash function takes an optional bound argument. Since our underlying hash tables don’t use bound argument, we actually wrap the internal hash function to allow the optional bound argument.

hash-table-ref  ht  key :optional thunk  [Function]

[SRFI-69]{srfi.69} Looks up the value corresponding to key in a hash table ht. If there’s no entry for key, thunk is called without arguments, and its result is returned. The default of thunk is to signal an error.

This convention differs from Gauche’s built-in hash-table-get, which takes optional default argument to be returned when the key doesn’t exist (see Section 6.14.1 [Hashtables], page 206). The following equivalence holds:

(hash-table-get ht key default) ≡ (hash-table-ref ht key (^[[] default))

hash-table-ref/default  ht  key  default  [Function]

[SRFI-69]{srfi.69} Looks up the value corresponding to key in a hash table ht. If the key doesn’t exist, default is returned. This is like Gauche’s hash-table-get, but default can’t be omitted (see Section 6.14.1 [Hashtables], page 206).

hash-table-set!  ht  key  val  [Function]

[SRFI-69]{srfi.69} This is the same as Gauche’s hash-table-put! (see Section 6.14.1 [Hashtables], page 206).

hash-table-update!  ht  key  proc :optional thunk  [Function]
hash-table-update!/default  ht  key  proc  default  [Function]

[SRFI-69]{srfi.69} Calls proc with the value associated to key in ht, and replace the value in ht with the result of proc. The following functional equivalences hold, though these procedures may be more efficient.

(hash-table-update! ht key proc thunk)
≡ (hash-table-set! ht key (proc (hash-table-ref ht key thunk)))

(hash-table-update!/default ht key proc default)
≡ (hash-table-set! ht key (proc (hash-table-ref/default ht key default)))

Note that Gauche’s built-in hash-table-update! is different, for it takes optional default argument instead of thunk (see Section 6.14.1 [Hashtables], page 206).

hash-table-size  ht  [Function]

[SRFI-69]{srfi.69} Returns the number of entries in a hash table ht. The same as Gauche’s hash-table-num-entries.

hash-table-walk  ht  proc  [Function]

[SRFI-69]{srfi.69} For each entry in a hash table ht, calls proc with two arguments, a key and its value. It’s the same as Gauche’s hash-table-for-each.
Chapter 11: Library modules - SRFIs

hash-table-merge! \(ht1 \ ht2\) [Function][SRFI-69]{srfi.69} Add all entries in a hash table \(ht2\) into a hash table \(ht1\), and returns \(ht1\).

hash \(obj\) :optional \(bound\) [Function][SRFI-69]{srfi.69} Like Gauche’s hash, except this one can take bound argument; if provided, it must be a positive integer, and the return value is limited between 0 and \((- bound 1)\), inclusive.

string-hash \(obj\) :optional \(bound\) [Function][SRFI-69]{srfi.69} These are like SRFI-13’s (see Section 11.6 [String library], page 692), except these don’t take start and end argument.

string-ci-hash \(obj\) :optional \(bound\) [Function][SRFI-69]{srfi.69} These are like SRFI-13’s (see Section 11.6 [String library], page 692), except these don’t take start and end argument.

hash-by-identity \(obj\) :optional \(bound\) [Function][SRFI-69]{srfi.69} This is Gauche’s eq-hash, except this one can take bound argument.

11.18 srfi.74 - Octet-addressed binary blocks

srfi.74 [Module] This module provides procedures to deal with blob, or a sequence of octets. In Gauche, a blob is simply an u8vector.

Most functionalities of this module is available in binary.io module (see Section 12.1 [Binary I/O], page 808), and in fact this module is a thin wrapper to it. We provide this module for the compatibility. If you’re writing Gauche-specific code, we recommend to use binary.io directly.

dendianness \(e\) [Macro][SRFI-74]{srfi.74} The argument \(e\) must be either big, little, or native. It expands to the implementation-specific endianness designator. In Gauche, the result is one of the symbols; see Section 6.3.7 [Endianness], page 138, for the details.

make-blob \(size\) [Function][SRFI-74]{srfi.74} Returns a freshly created blob that can hold \(size\) octets. In Gauche, this is the same as (make-u8vector \(size\)).

blob? \(obj\) [Function][SRFI-74]{srfi.74} Returns #t if \(obj\) is a blob, #f otherwise. In Gauche, this is the same as (u8vector? \(obj\)).

blob-length \(blob\) [Function][SRFI-74]{srfi.74} Returns the size of the blob, in octets. In Gauche, this is the same as (u8vector-length \(blob\)).

blob-uint-ref \(size\ \) \(endian\ \) \(blob\) \(pos\) [Function][SRFI-74]{srfi.74} Read an unsigned integer of \(size\) octets beginning at the position of \(pos\) from \(blob\), respectively.

blob-sint-ref \(size\ \) \(endian\ \) \(blob\) \(pos\) [Function][SRFI-74]{srfi.74} Read a signed integer of \(size\) octets beginning at the position of \(pos\) from \(blob\), respectively.

These are wrappers of (get-uint \(size\ \) \(blob\) \(pos\ \) \(endian\)) and (get-sint \(size\ \) \(blob\) \(pos\ \) \(endian\)) in binary.io module (see Section 12.1 [Binary I/O], page 808), except that blob-uint-ref/blob-sint-ref only accept u8vector as \(blob\).
blob-uint-set! size endian blob pos val  [Function]
blob-sint-set! size endian blob pos val  [Function]

[SRFI-74]{srfi.74} Store an unsigned or signed integer val of size octets into blob starting at the position of pos, respectively.

These are wrappers of (put-uint! size blob pos val endian) and (put-sint! size blob pos val endian) in binary.io module (see Section 12.1 [Binary I/O], page 808), except that blob-uint-set!/blob-sint-set! only accept u8vector as blob.

blob-u8-ref blob pos  [Function]
blob-u8-set! blob pos val  [Function]
blob-s8-ref blob pos  [Function]
blob-s8-set! blob pos val  [Function]

[SRFI-74]{srfi.74} Get/set an unsigned or signed integer as a octet at pos from/to blob.

These are wrappers of get-u8, put-u8!, get-s8 and put-s8! in binary.io, respectively.

blob-u16-ref endian blob pos  [Function]
blob-u16-set! endian blob pos val  [Function]
blob-s16-ref endian blob pos  [Function]
blob-s16-set! endian blob pos val  [Function]
blob-u32-ref endian blob pos  [Function]
blob-u32-set! endian blob pos val  [Function]
blob-s32-ref endian blob pos  [Function]
blob-s32-set! endian blob pos val  [Function]
blob-u64-ref endian blob pos  [Function]
blob-u64-set! endian blob pos val  [Function]
blob-s64-ref endian blob pos  [Function]
blob-s64-set! endian blob pos val  [Function]

[SRFI-74]{srfi.74} Get/set an unsigned or signed integer of the indicated length at pos from/to blob, using the specified endian.

These are wrappers of corresponding get-XX and put-XX! in binary.io; note that the argument orders differ, though.

blob-u16-native-ref blob pos  [Function]
blob-u16-native-set! blob pos val  [Function]
blob-s16-native-ref blob pos  [Function]
blob-s16-native-set! blob pos val  [Function]
blob-u32-native-ref blob pos  [Function]
blob-u32-native-set! blob pos val  [Function]
blob-s32-native-ref blob pos  [Function]
blob-s32-native-set! blob pos val  [Function]
blob-u64-native-ref blob pos  [Function]
blob-u64-native-set! blob pos val  [Function]
blob-s64-native-ref blob pos  [Function]
blob-s64-native-set! blob pos val  [Function]

[SRFI-74]{srfi.74} Get/set an unsigned or signed integer of the indicated length at pos from/to blob, using the native endianness.

These are wrappers of corresponding get-XX and put-XX! in binary.io; note that the argument orders differ, though.

blob=? blob1 blob2  [Function]

[SRFI-74]{srfi.74} This is the same as u8vector=? in gauche.uvector.
### Chapter 11: Library modules - SRFIs

**SRFI-74**

**blob-copy!**  src sstart target tstart n  
[Function]

[SRFI-74]  Returns a fresh copy of blob. The same as u8vector-copy in gauche.uvector.

**blob-copy**  blob  
[Function]

**blob->u8-list**  size endian blob  
[Function]

**uint-list->blob**  size endian list  
[Function]

**sint-list->blob**  size endian list  
[Function]

**check**  expr => expected  
[Macro]

**check**  expr (=> equal) expected  
[Macro]

**check-ec**  qualifier ... expr => expected (argument ...)  
[Macro]

**check-ec**  qualifier ... expr (=> equal) expected (argument ...)  
[Macro]

**SRFI-78**  - Lightweight testing

This srfi defines check and check-ec macro, along with a few helper procedures, to write tests. Especially, check-ec uses the same comprehension style as SRFI-42 to run test expressions with various combinations of input easily.

We implemented this module to work with gauche.test (see Section 9.34 [Unit testing], page 521). Specifically, if the check macros are invoked while gauche.test is active (that is, between test-start and test-end), the check macros are simply a wrapper of Gauche’s test macro; the results are tracked and reported by gauche.test.

If this module is used without gauche.test, however, it works as specified by the srfi, including reporting. So, a tests using this srfi can be run in both ways—if it is loaded by itself, it runs as vanilla SRFI-78, and if it is included in a test file that uses gauche.test, it runs as a part of gauche.test.

**check**  expr => expected  
[Macro]

**check**  expr (=> equal) expected  
[Macro]

**check-ec**  qualifier ... expr => expected (argument ...)  
[Macro]

**check-ec**  qualifier ... expr (=> equal) expected (argument ...)  
[Macro]

**check-ec**  qualifier ... expr => expected  
[Macro]
check-ec qualifier ... expr (=> equal) expected [Macro]
[SRFI-78]{srfi.78} Evaluates expr, equal and expected repeatedly in the environment where the qualifiers bind variables, and each time the results of expr and equal are compared with equal. If the results don’t agree, the failure is recorded with the given argument ..., which can be useful to diagnose which combination of bindings the test failed on. Single failure stops the iteration. If the results of expr and expected agrees on all the iterations, the entire check-ec is regarded as success.

The way qualifiers work is the same as SRFI-42. See Section 11.11 [Eager comprehensions], page 711, for the details.

The following example tests Fermat numbers \( 2^{2^n} + 1 \) are primes. It fails since \( F_5 \) is a composite.

```
(use math.prime)
(check-ec (: n 6)
   (:let fn (+ (expt 2 (expt 2 n)) 1))
   (bpsw-prime? fn)
 => #t (n))
```

prints Checking (bpsw-prime? fn), expecting #t => ERROR: got #f, with n: 5

The entire check-ec form is treated as one check for the sake of reporting.

cHECK-report [Function]
[SRFI-78]{srfi.78} If this module is used stand-alone, prints the summary of test results to the current output port.

If this module is running inside gauche.test, this does nothing. Reporting is done by gauche.test.

cHECK-set-mode! mode [Function]
[SRFI-78]{srfi.78} Sets how the test progress and result will be reported. This only has an effect when check is run without gauche.test.

Valid mode is one of the following symbols:

- off Do not report the result at all, even from (check-report).
- summary Report the summary of the results when (check-report) is called.
- report-failed In addition to summary, report any failed checks as they occur.
- report In addition to summary, report each result of check as they occur.

The default is report.

cHECK-reset! [Function]
[SRFI-78]{srfi.78} Clears the internal state to keep track of number of performed checks and failed checks, etc. This doesn’t affect gauche.test bookkeeping.

cHECK-passed? expected-total-count [Function]
[SRFI-78]{srfi.78} This can be used to programatically query whether the expected number of tests are passed since the last check-reset! or the beginning. The expected-total-count must be a number, and it returns #t iff the internal count of passed test matches it and there is no failed checks.

This works even if checks are run with gauche.test. However, only the tests using check and check-ec are counted.
11.20 srfi.98 - Accessing environment variables

srfi.98 [Module]
This srfi defines a portable way to access the underlying system’s environment variables. Gauche supports such procedures built-in (see Section 6.24.3 [Environment inquiry], page 295), but portable programs may want to use srfi API instead.

get-environment-variable name [Function]
[SRFI-98] Returns a string value of an environment variable named by a string name. If the named environment variable doesn’t exist, #f is returned.

This is equivalent to sys-getenv.

\[
(get\text{-}environment\text{-variable} \ "PATH") \Rightarrow \="/bin:/usr/sbin:/usr/bin"
\]

get-environment-variables [Function]
[SRFI-98] Returns an assoc list of the name and the value of each environment variable.

This is equivalent to sys-environ->alist without the optional argument.

\[
(get\text{-}environment\text{-variables}) \Rightarrow \(("PATH" \ . \="/bin:/usr/sbin:/usr/bin")
\ldots\)
\]

11.21 srfi.101 - Purely functional random-access pairs and lists

srfi.101 [Module]
SRFI-101 has become a part of R7RS large. See Section 10.3.9 [R7RS random-access lists], page 622.

Special treatment of cond-expand: The feature conditional cond-expand implicitly makes the checked library loaded if available. That is, you can usually say (cond-expand (srfi-N <code>)) where <code> can assume SRFI-N is already available, without saying (use srfi.N).

However, srfi.101 exports the names that conflicts with standard primitive pair and list operators, and importing it without prefix or renaming would surprise you. To avoid confusion, we don’t automatically use srfi.101 even if it is used as feature conditional.

The recommended way is to use library feature requirements, and use srfi.101 with your desired prefix, e.g. (cond-expand ((library srfi.101) (use srfi.101 :prefix ra:))).

11.22 srfi.106 - Basic socket interface

srfi.106 [Module]
A portable basic socket interface.

Although comprehensive network API is provided by gauche.net (see Section 9.22 [Networking], page 464), it is Gauche-specific. This srfi provides a small subset of socket operations, but it offers a portable way to create applications that needs simple networking.

Note that some procedures have the same name as the ones in gauche.net, but the interface may differ.

A socket object created by this srfi’s API is an instance of Gauche’s <socket>, so it can be passed to the API in gauche.net and vice versa.
The following procedures are exactly the same as defined in `gauche.net`. See Section 9.22 [Networking], page 464, for the details.

- `socket-accept`
- `socket-shutdown`
- `socket-close`
- `socket-input-port`
- `socket-output-port`

### Socket object

**make-client-socket**

```scheme
node service :optional ai-family ai-socktype ai-flags ai-protocol
```

[SRFI-106]{srfi.106} Creates and returns a socket to communicate with the node `node` and service. If the socket type is connection-oriented (that is, `ai-socktype` is `*sock-stream*`, which is the default), the returned socket is already connected.

Both `node` and `service` must be strings. The `node` argument is passed to `getaddrinfo(3)` to resolve to the server IP address(es). A service name solely consists of decimal digits is interpreted as a port number.

The default value of optional arguments are as follows: `*af-inet*` for `ai-family`, `*sock-stream*` for `ai-socktype`, `(socket-merge-flags *ai-v4mapped* *ai-addrconfig*)` for `ai-flags`, and `*ipproto-ip*` for `ai-protocol`. See below for valid flag values.

This API differs from `make-client-socket` in `gauche.net`.

```scheme
(make-client-socket "127.0.0.1" "80")
⇒ a <socket> connected to port 80 of localhost
```

**make-server-socket**

```scheme
service :optional ai-family ai-socktype ai-protocol
```

[SRFI-106]{srfi.106} Creates and returns a server socket that binds and listens at the port specified by `service`, which must be a string. A service name solely consists of decimal digits is interpreted as a port number.

The default value of optional arguments are as follows: `*af-inet*` for `ai-family`, `*sock-stream*` for `ai-socktype`, and `*ipproto-ip*` for `ai-protocol`. See below for valid flag values.

This API differs from `make-server-socket` in `gauche.net`.

**socket?**

```scheme
obj
```

[SRFI-106]{srfi.106} Equivalent to `(is-a? obj <socket>)`.

### Communication

**socket-send**

```scheme
socket u8vector :optional flags
```

[SRFI-106]{srfi.106} Almost same as `socket-send` in `gauche.net`, except that this procedure only accepts a u8vector as the message. (The one in `gauche.net` can take a string as well.)

Returns the number of octets that are actually sent.

**socket-recv**

```scheme
socket size :optional flags
```

[SRFI-106]{srfi.106} This is like `socket-recv` in `gauche.net`, except that this procedure returns the received data in u8vector, instead of a string. If the peer has shut down the connection, this procedure returns an empty u8vector, `#u8()`.

The `size` argument specifies the maximum size of the receiving data. The returned vector may be shorter if that much data is received.
Chapter 11: Library modules - SRFIs

Flags
The srfi provides common names for constants of typical socket flags, as well as macros that map symbolic name(s) to the flags.

socket-merge-flags flag . . . [Function] [SRFI-106]{srfi.106} Merge bitwise flags. This is simply logior in Gauche.

socket-purge-flags base-flag flag . . . [Function] [SRFI-106]{srfi.106} Drop the bitwise flags in base-flag that are set in flag . . .

Address family
*af-inet* AF_INET
*af-inet6* AF_INET6
*af-unspec* AF_UNSPEC

address-family name [Macro] [SRFI-106]{srfi.106} Name can be either one of symbols inet, inet6, or unspec, and the macro expands into the value of *af-inet*, *af-inet6* or *af-unspec*, respectively.

If name is other object, an error is signaled.

Socket domain
*sock-stream* SOCK_STREAM
*sock-dgram* SOCK_DGRAM

socket-domain name [Macro] [SRFI-106]{srfi.106} Name can be either one of symbols stream or datagram, and the macro expands into the value of *sock-stream* and *sock-dgram*, respectively.

If name is other object, an error is signaled.

Address info
*ai-canonname* AI_CANONNAME
*ai-numerichost* AI_NUMERICHOST
*ai-v4mapped* AI_V4MAPPED
*ai-all* AI_ALL
*ai-addrconfig* AI_ADDRCONFIG

address-info name . . . [Macro] [SRFI-106]{srfi.106} Maps combination of names canoname, numerichost, v4mapped, all and addrconfig to the combination of corresponding flags.

An error is signaled if other symbols are passed. (Note: canoname for *ai-canonname*).

Protocol
*ipproto-ip* IPPROTO_IP
*ipproto-tcp* IPPROTO_TCP
*ipproto-udp* IPPROTO_UDP

ip-protocol name [Macro] [SRFI-106]{srfi.106} Maps one of names ip, tcp, and udp to the corresponding flag value.

An error is signaled if other symbol is passed.
Chapter 11: Library modules - SRFIs

Message type

*msg-none* 0
*msg-peek* MSG_PEEK
*msg-oob* MSG_OOB
*msg-waitall* MSG_WAITALL

message-type name ... [Macro]
[SRFI-106]{srfi.106} Maps combination of names none, peek, oob and wait-all to the combination of corresponding flags.
An error is signaled if other symbols are passed. (Note: wait-all for *msg-waitall*).

Shutdown method

*shut-rd* SHUT_RD
*shut-wr* SHUT_WR
*shut-rdwr* SHUT_RDWR

shutdown-method name ... [Macro]
[SRFI-106]{srfi.106} Maps combination of names read and write to the combination of corresponding flags.
An error is signaled if other symbols are passed.

11.23 srfi.112 - Environment inquiry

srfi.112 [Module]
This srfi provides a portable way to obtain runtime information.

implementation-name [Function]
[SRFI-112]{srfi.112} The name of the implementation. Returns a string "Gauche".

implementation-version [Function]
[SRFI-112]{srfi.112} Returns a string of Gauche’s version. The same as gauche-version (see Section 6.24.3 [Environment inquiry], page 295).

cpu-architecture [Function]
[SRFI-112]{srfi.112} Returns a string of CPU architecture info, such as "x86_64". Same as the machine field of the return value of sys-uname (see Section 6.24.8 [System inquiry], page 313).

machine-name [Function]
[SRFI-112]{srfi.112} Returns the host name. Same as the nodename field of the return value of sys-uname. (see Section 6.24.8 [System inquiry], page 313).

os-name [Function]
[SRFI-112]{srfi.112} Returns the OS name. Same as the sysname field of the return value of sys-uname.

os-version [Function]
[SRFI-112]{srfi.112} Returns the OS version. Same as the release field of the return value of sys-uname.
Here’s an example of output. It is likely to differ on your environment.

```scheme
(gosh> (implementation-name)  
"Gauche"
(gosh> (implementation-version)  
"0.9.5"
(gosh> (cpu-architecture)  
"x86_64"
(gosh> (machine-name)  
"scherzo"
(gosh> (os-name)  
"Linux"
(gosh> (os-version)  
"3.2.0-89-generic"

11.24 srfi.114 - Comparators

srfi.114 [Module]
This module is provided for the compatibility of code using SRFI-114. The new code should use SRFI-128, which is fully built-in.

Note that SRFI-114’s `make-comparator` has different interface from built-in (SRFI-128) `make-comparator`. If you simply say `(use srfi.114)` or `(import (srfi 114))`, it shadows the built-in one. If you need to use some of SRFI-114 procedures, it’s better to import them selectively.

Note that comparators created with SRFI-114 procedures are the same type as the ones created with SRFI-128 procedures and can be used interchangeably.

The following procedures are built-in. See Section 6.2.4 [Basic comparators], page 116, for the detailed documentation. Those are also exported from srfi.114 for the compatibility.

Predicates `comparator?`,

Standard comparators
```scheme
  boolean-comparator,  char-comparator,  char-ci-comparator,
  string-comparator,  string-ci-comparator,  symbol-comparator,
  exact-integer-comparator,  integer-comparator,  rational-comparator,
  real-comparator,  complex-comparator,  number-comparator,
  pair-comparator,  list-comparator,  vector-comparator,
  bytevector-comparator,  uvector-comparator
```

The default comparator
```scheme
  default-comparator
```

Wrapped equality predicates
```scheme
  eq-comparator,  eqv-comparator,  equal-comparator
```

Accessors `comparator-equality-predicate`, `comparator-comparison-procedure`, `comparator-hash-function`

Primitive applicators
```scheme
  comparator-test-type,  comparator-check-type,  comparator-compare,
  comparator-hash
```

Comparison predicates
```scheme
  =?,  <?,  <=?,  >?,  >=?
```
Basic comparator interface

**make-comparator** type-test equal compare hash :optional name

[SRFI-114]{srfi.114} This is SRFI-114 style comparator constructor. The optional name argument is Gauche’s extension.

This is the same as built-in **make-comparator/compare**. See Section 6.2.4 [Basic comparators], page 116, for the details.

Do not confuse this with built-in (SRFI-128) **make-comparator**; if you (use srfi.114), this one shadows the built-in one.

Note that a comparator works for both SRFI-114 and SRFI-128 procedures, regardless of how it is constructed.

**comparator-comparison-procedure?** c

**comparator-hash-function?** c

[SRFI-114]{srfi.114} Returns true iff a comparator c can be used to order objects or to hash them, respectively. These are aliases of built-in comparator-ordered? and comparator-hashable?.

**comparator-type-test-procedure** c

[SRFI-114]{srfi.114} Returns type test predicate of a comparator c. This is an alias of built-in comparator-type-test-predicate.

**comparator-equal?** c a b

[SRFI-114]{srfi.114} Checks equality of a and b using the equality predicate of a comparator c. This can be also written in =?, which is built-in (see Section 6.2.4.2 [Comparator predicates and accessors], page 118).

(=? c a b)

Auxiliary comparator constructors

**make-inexact-real-comparator** epsilon rounding nan-handling

[SRFI-114]{srfi.114} Returns a comparator for inexact real numbers, taking into account of errors and NaNs.

The basic idea is that we compare two finite real numbers after rounding them to epsilon interval, which must be a nonnegative real number. (Note that it’s not to compare two numbers “close enough”, as often being done to compare inexact numbers. “Close enough” scheme won’t be transitive.)

The rounding mode is specified by the rounding argument. It can be either one of the symbols round, ceiling, floor or truncate, or a procedure that takes two arguments, a real number and an epsilon, and returns the rounded result of the first argument according to the given epsilon.

The nan-handling argument determines how to handle the case to compare NaN and non-NaN numbers. (If both are NaNs, this comparator regards them as equal). It can be either one of the followings:

min If it’s a symbol min, NaN is compared as smaller than all other real numbers, even than -inf.0.

max If it’s a symbol min, NaN is compared as greater than all other real numbers, even than +inf.0.

error If it’s a symbol error, an error is signaled.
a procedure taking one argument

The procedure is invoked with the real number which is not NaN. If it ever returns, it must return either 1, 0 or -1, for it’s used as the result of the comparison procedure of the comparator. However, since the procedure doesn’t know which argument is non-NaN, it’s hard to have consistent semantics; the best bet is to throw a custom error.

```
(define c (make-inexact-real-comparator 0.1 'round 'error))
```

```
(comparator-compare c 0.112 0.098) ⇒ 0
(comparator-compare c 0.131 0.172) ⇒ -1
```

Note: Rounding to the nearest epsilon interval would involve scaling inexact numbers, and that may reveal small difference between the actual number and its notation. For example, an inexact real number denoted as 0.15 is actually slightly smaller than 15/100, and rounding with epsilon 0.1 would result 0.1, not 0.2.

**Function**

`make-car-comparator cmpr`

`make-cdr-comparator cmpr`

[SRFI-114] Returns comparators that accept pairs, and compare them with their car or cdr by `cmpr`, respectively.

Using `make-key-comparator`, these can be written as follows (see Section 6.2.4.4 [Combining comparators], page 121, for `make-key-comparator`).

```
(define (make-car-comparator cmpr)
  (make-key-comparator cmpr pair? car))
```

```
(define (make-cdr-comparator cmpr)
  (make-key-comparator cmpr pair? cdr))
```

**Function**

`make-list-comparator element-comparator`

`make-vector-comparator element-comparator`

`make-bytevector-comparator element-comparator`

[SRFI-114] Returns a new comparator that compares lists, vectors and bytevectors element-wise using `element-comparator`, respectively. These are more general versions of `list-comparator`, `vector-comparator` and `bytevector-comparator`, which use `default-comparator` as `element-comparator`.

For a list comparator, it is an error to pass improper lists.

Note that comparing sequences of different lengths is slightly different between lists and vector/bytevectors. List comparator uses “dictionary” order, so (1 3) comes after (1 2 3), assuming elements are compared numerically. For vectors and bytevectors, shorter one always precedes the other, so #(1 3) comes before #(1 2 3).

**Function**

`make-listwise-comparator type-test element-comparator empty? head tail`

[SRFI-114] More general version of `make-list-comparator`. Returns a comparator that compares structures which can be traversed using three procedures, `empty?`, `head` and `tail`. Each of those procedure receives a structure to be compared, and `empty?` must return #t if the structure is empty, `head` must return the first element in the structure, and `tail` must return the same type of structure containing all the elements but the head. The `type-test` predicate checks if the arguments passed to the comparator to be a suitable structure.

That is, `make-list-comparator` can be written in `make-listwise-comparator` as follows.

```
(make-list-comparator element-comparator)
```
\[(\text{make-listwise-comparator list? element-comparator null? car cdr})\]

This can be used to compare list-like structures. For example, the following call returns a comparator that compares elements of two lazy streams (see Section 12.90 [Stream library], page 1033).

\[(\text{make-listwise-comparator stream? element-comparator stream-null? stream-car stream-cdr})\]

\textbf{make-vectorwise-comparator type-test element-comparator length ref} \ [Function]

[SRFI-114]{srfi.114} More general version of \text{make-vector-comparator}. Returns a comparator that compares structures which can be traversed using two procedures, \text{length} and \text{ref}. The \text{length} procedure must return the number of elements in the structure. The \text{ref} procedure receives a structure and a nonnegative exact integer index \(k\), and must return \(k\)-th element of the structure.

That is, the following equivalence holds:

\[(\text{make-vector-comparator element-comparator})\]
\[\equiv\]
\[(\text{make-vectorwise-comparator vector? element-comparator vector-length vector-ref})\]

\[(\text{make-bytevector-comparator element-comparator})\]
\[\equiv\]
\[(\text{make-vectorwise-comparator u8vector? element-comparator u8vector-length u8vector-ref})\]

\textbf{make-pair-comparator car-comparator cdr-comparator} \ [Function]

[SRFI-114]{srfi.114} Creates a comparator that compares pairs, with their cars by \text{car-comparator} and their cdrs by \text{cdr-comparator}.

\textbf{make-improper-list-comparator element-comparator} \ [Function]

[SRFI-114]{srfi.114} This may be understood as recursive pair comparator; if objects to be compared are pairs, we recurse their cars then their cdrs. If objects to be compared are not pairs, we use \text{element-comparator} to compare them.

\textbf{make-selecting-comparator comparator1 comparator2 ...} \ [Function]

[SRFI-114]{srfi.114} This creates a comparator that works any one of the given comparators; the objects to be compared are type-tested with each of the comparators in order, and the first comparator that accepts all objects will be used.

\textbf{make-refining-comparator comparator1 comparator2 ...} \ [Function]

[SRFI-114]{srfi.114} This is similar to \text{make-selecting-comparator}, except that if the first comparator that accepts given objects to compare finds they are equal (or 0 by the comparison procedure), it tries other comparators down the list, if any.

\textbf{make-reverse-comparator comparator} \ [Function]

[SRFI-114]{srfi.114} Returns a comparator that just reverses the comparison order of \text{comparator}.

\textbf{make-debug-comparator comparator} \ [Function]

[SRFI-114]{srfi.114}
Comparison procedure constructors

```scheme
(make-comparison< lt-pred)  ; [Function]
(make-comparison> gt-pred)  ; [Function]
(make-comparison<= le-pred) ; [Function]
(make-comparison>= ge-pred) ; [Function]
(make-comparison=< eq-pred lt-pred) ; [Function]
(make-comparison=> eq-pred gt-pred) ; [Function]
```

[SRFI-114]{srfi.114} Utility procedures to create a comparison procedure (the one returns -1, 0, or 1) from the given predicate. For example, `make-comparison<` can be defined as follows:

```scheme
(define (make-comparison< pred)
  (\[a b\] (cond [(pred a b) -1]
                 [(pred b a) 1]
                 [else 0])))
```

Comparison syntax

```scheme
if3 expr less equal greater          ; [Macro]
[SRFI-114]{srfi.114} Three-way if: Evaluates expr, and then evaluates either one of less, equal, or greater, depending on the value of expr is either less than zero, equal to zero, or greater than zero, respectively.
```

```scheme
(if=? expr consequent :optional alternate) ; [Macro]
(if<? expr consequent :optional alternate) ; [Macro]
(if>? expr consequent :optional alternate) ; [Macro]
(if<=? expr consequent :optional alternate) ; [Macro]
(if>=? expr consequent :optional alternate) ; [Macro]
(if-not=? expr consequent :optional alternate) ; [Macro]
```

[SRFI-114]{srfi.114} Conditional evaluation according to comparison expression expr; that is, `if OP?` evaluates consequent if `(OP expr 0)` is true, otherwise it evaluates alternate when provided.

```scheme
(if<? (compare 10 20) 'yes) ⇒ yes
(if>=? (compare 10 20) 'yes 'no) ⇒ no
```

Comparison predicate constructors

```scheme
(make=? comparator) ; [Function]
(make<? comparator) ; [Function]
(make>? comparator) ; [Function]
(make<=? comparator) ; [Function]
(make=>? comparator) ; [Function]
```

[SRFI-114]{srfi.114}

```scheme
((make=? comparator) obj1 obj2 obj3 ...) ≡ (=? comparator obj1 obj2 obj3 ...)
```

Interval comparison predicates

```scheme
(in-open-interval? [comparator] obj1 obj2 obj3) ; [Function]
in-closed-interval? [comparator] obj1 obj2 obj3 ; [Function]
in-open-closed-interval? [comparator] obj1 obj2 obj3 ; [Function]
in-closed-open-interval? [comparator] obj1 obj2 obj3 ; [Function]
```

[SRFI-114]{srfi.114} Check if `obj1`, `obj2` and `obj3` has the following relationships:

```scheme
(and (op1 obj1 obj2) (op2 obj2 obj3))
```
Where each of \texttt{op1} and \texttt{op2} can be \texttt{(make<? comparator)} (if that end is \textit{open}), or \texttt{(make<=? comparator)} (if that end is \textit{closed}).

When \texttt{comparator} is omitted, the default comparator is used.

\begin{verbatim}
(use srfi.42)
(list-ec (: x 0 5) (list x (in-closed-open-interval? 1 x 3)))
⇒ ((0 #f) (1 #t) (2 #t) (3 #f) (4 #f))
\end{verbatim}

Min/max comparison procedures

\texttt{comparator-min} and \texttt{comparator-max} are the same as \texttt{srfi.162}. See Section 11.32 [Comparator sublibrary], page 744.

\section*{11.25 srfi.118 - Simple adjustable-size strings}

\texttt{srfi.118} [Module]

This SRFI defines two string mutating operations that can change the length of the string: \texttt{string-append!} and \texttt{string-replace!}.

Note that, in Gauche, the body of strings is immutable; when you mutate a string, Gauche creates a fresh new string body and just switch a pointer in the original string to point to the new string body. So it is not a problem to implement this SRFI in Gauche, but it also means you won’t get any performance benefit by using these operations. Using immutable counterparts (\texttt{string-append} and \texttt{string-replace}) gives you the same performance. (Be aware that the interface is slightly different from the immutable versions.)

We provide this module only for the compatibility. Gauche-specific programs should stay away from this module. Particularly, avoid code like the example in SRFI-118 document (build a string by \texttt{append!}-ing small chunks at a time)—they’re quadratic on Gauche.

\texttt{string-append!} \textit{string values} . . . [Function]

\begin{verbatim}
|SRFI-118|{srfi.118} The \texttt{string} argument must be a mutable string. Modify \texttt{string} by appending \texttt{values}, each of which is either a character or a string.

\begin{verbatim}
(rlet1 a (string-copy "abc")
  (string-append! a #\X "YZ")
⇒ "abcXYZ"
\end{verbatim}
\end{verbatim}

\texttt{string-replace!} \textit{dst dst-start dst-end src} :optional \textit{src-start src-end} [Function]

\begin{verbatim}
|SRFI-118|{srfi.118} The \texttt{dst} argument must be a mutable string. Replace \texttt{dst} between \texttt{dst-start} (inclusive) and \texttt{dst-end} (exclusive) with a string \texttt{src}. The optional arguments \texttt{src-start} and \texttt{src-end} limits the region of \texttt{src} to be used.

Be aware that the order of arguments differ from SRFI-13’s \texttt{string-replace} (see Section 11.6.12 [SRFI-13 Other string operations], page 700); \texttt{string-replace!} resembles to \texttt{string-copy!} (also in SRFI-13), rather than \texttt{string-replace}.

\begin{verbatim}
(rlet1 a (string-copy "abc")
  (string-replace! a 1 2 "XYZ")
⇒ "aXYZc"
\end{verbatim}
\end{verbatim}

\section*{11.26 srfi.120 - Timer APIs}

\texttt{srfi.120} [Module]

This \texttt{srfi} provides a device to run tasks in a specified time.

In Gauche, this is a thin wrapper of \texttt{control.scheduler} module (see Section 12.11 [Scheduler], page 824, for the details). Use this module if you need portability.
Chapter 11: Library modules - SRFIs

make-timer :optional error-handler
[Function]
[SRFI-120]{srfi.120} Create a new timer, with error-handler as an error handler. The error-handler argument must be #f or a procedure that takes one argument. If it is a procedure, it is called when the task thunk raises an error, with the raised condition as an argument. Its return value is ignored. If omitted, #f is assumed.

In Gauche, returned timer object is just an instance of <scheduler>, and error-handler is set in its error-handler slot. See Section 12.11 [Scheduler], page 824, for the details.

timer? obj
[Function]
[SRFI-120]{srfi.120} Returns #t iff obj is a timer (an instance of <scheduler>).

Same as scheduler? in control.scheduler (see Section 12.11 [Scheduler], page 824).

timer-cancel! timer
[Function]
[SRFI-120]{srfi.120} Stops the timer. No tasks in the queue will be executed, and no new tasks is accepted. Once timer is canceled, it can’t be restarted.

If the timer’s tasks ever raised an error, and either the timer doesn’t have an error handler, or an error handler itself raises an error, then the condition object is kept in the timer and reraised from this procedure.

See also scheduler-terminate! in control.scheduler (see Section 12.11 [Scheduler], page 824).

timer-schedule! timer thunk when :optional period
[Function]
[SRFI-120]{srfi.120} Creates a new task to run thunk in the timer, at when. The when argument specifies the relative time from the moment this procedure is called. It must be either an integer as the number of milliseconds, or a time delta object created by make-timer-delta.

If the period argument is given and non zero, the task will be repeated in every period. It must be a nonnegative integer in milliseconds or a time delta object. If it is zero, task is non-repeating.

Returns a task-id, which can be used to reschedule or remove the task later. The srfi specifies a task id to be any printable Scheme object; Gauche uses an integer.

Note: In Gauche, a time delta object is a SRFI-19 time-duration time (see Section 11.7.1 [SRFI-19 Time types], page 701). This is implemented on top of scheduler-schedule! in control.scheduler (see Section 12.11 [Scheduler], page 824), but note that the unit of when and period is different from it.

timer-reschedule! timer task-id when :optional period
[Function]
[SRFI-120]{srfi.120} Change the timing when task identified by task-id in timer is executed. The task-id should be the object returned from previous timer-schedule! on the same timer.

The semantics of when and period are the same as timer-schedule!.

If there’s no task with task-id in timer (including the case that the task was one-shot and already run), an error is thrown. (The behavior is undefined in the srfi).

This procedure returns task-id.

This is implemented on top of scheduler-reschedule! in control.scheduler (see Section 12.11 [Scheduler], page 824), but note that the unit of when and period is different from it.

timer-task-remove! timer task-id
[Function]
[SRFI-120]{srfi.120} Removes the task specified by task-id from the timer. Returns #t if the task is actually removed, #f if the timer doesn’t have the specified task.
**timer-task-exists?** timer task-id

[Function]

[SRFI-120]{srfi.120} Returns #t iff the timer has a task specified by task-id. A non-repeated task is automatically removed from timer once executed.

**make-timer-delta n unit**

[Function]

[SRFI-120]{srfi.120} A constructor for time-delta object. It can be passed to when and period arguments of timer-schedule! and timer-reschedule!.

In Gauche, a time-delta object is an instance of `<time>` with time-duration type (see Section 11.7.1 [SRFI-19 Time types], page 701). The srfi doesn’t specify the concrete implementation, though, so portable code should use this procedure.

n is an integer, and unit must be one of the following symbols: h (hour), m (minute), s (second), ms (millisecond), us (microsecond) or ns (nanosecond).

**timer-delta? obj**

[Function]

[SRFI-120]{srfi.120} Returns #t iff obj is a time-delta object, which is, in Gauche, an instance of `<time>` object with time-duration type.

### 11.27 srfi.129 - Titlecase procedures

**srfi.129**

This srfi defines 3 procedures handling titlecase of characters. All of the procedures are supported by Gauche, and this module serves as a compatibility interface.

These two procedures are built-in (see Section 6.9 [Characters], page 160):

- char-title-case?
- char-titlecase

The following procedure is provided in gauche.unicode (see Section 9.37.3 [Full string case conversion], page 554).

- string-titlecase

### 11.28 srfi.130 - Cursor-based string library

**srfi.130**

This is an reduced version of srfi.13 that supports cursors in addition to indexes. The consistency with R7RS and recent srfis are also considered.

Gauche supports string cursors natively (see Section 6.11.5 [String cursors], page 174), so all built-in and SRFI-13 string procedures that takes string indexes can als otake string cursors as well. You want to use this module explicitly, though, if you’re writing a portable code.

The following are built-in. See Section 6.11.5 [String cursors], page 174, for the details.

- string-cursor?
- string-cursor-start
- string-cursor-end
- string-cursor-next
- string-cursor-prev
- string-cursor-forward
- string-cursor-back
- string-cursor=?
- string-cursor<=?
- string-cursor>?
- string-cursor=>?
- string-cursor-diff
- string-cursor->index
- string-index->cursor

The following procedures are simply aliases of the builtin version without the /cursor or /cursors suffix.

- string->list/cursors
- string->vector/cursors
- string-copy/cursors
- substring/cursors

The following procedures are defined in SRFI-13. See Section 11.6 [String library], page 692, for the details. Note: Some of those procedures in SRFI-13 accept only indexes. In SRFI-130, both indexes and cursors are accepted. Our SRFI-13 implementation shares the same
procedure with SRFI-130, so they accept both indexes and cursors but such code won’t be portable as SRFI-13.

```
string-null?  string-any  string-every  string-tabulate
string-unfold string-unfold-right  reverse-list->string
string-take  string-drop  string-take-right  string-drop-right
string-pad  string-pad-right  string-trim  string-trim-right
string-trim-both  string-replace  string-prefix-length
string-suffix-length  string-prefix?  string-suffix
string-concatenate string-concatenate-reverse  string-count
string-filter  string-reverse
```

The following procedures are the same as SRFI-152, except that the arguments that takes integer indexes can also accept string cursors (see Section 11.29 [String library (reduced)], page 740):

```
string-replicate  string-remove  string-split
```

The following procedures are also defined in SRFI-13, but SRFI-130 redefines them to return a string cursor instead of an integer index. They are described below.

```
string-index  string-index-right
string-skip  string-skip-right
string-contains  string-contains-right
```

```
string-index string pred :optional start end  [Function]
string-index-right string pred :optional start end  [Function]
string-skip  string pred :optional start end  [Function]
string-skip-right string pred :optional start end  [Function]
```

[SRFI-130] {srfi.130} Find the position of a character that satisfies pred (string-index/string-index-right), or does not satisfy pred (string-skip/string-skip-right), in a string. The character is searched from left to right in string-index/string-skip, and right to left in string-index-right/string-skip-right.

While SRFI-13’s procedures of the same names returns integer index, these returns a string cursor instead.

The returned cursor from string-index/string-skip points the leftmost character that satisfies/does not satisfy pred, while the returned cursor from string-index-right points the next (right) character of the rightmost one that satisfies/does not satisfy pred. If there’re no characters that satisfy pred, string-index returns a cursor at the end of the string, while string-index-right returns a cursor at the beginning of the string. (The *-skip variation reverses the condition).

(This is easier to understand if you think the cursor is between characters; if we scan a string from left to right, the cursor looks at the character on its right; if we scan from right to left, the cursor looks at the character on its left.)

Be aware that they still return a cursor even no characters satisfy the condition, while SRFI-13 versions return #f in such cases.

The optional start/end can be string cursors or integer indexes to limit the region of string to search.

```
(define *s* "abc def ghi")
```

```
(string-cursor->index *s* (string-index *s* char-whitespace?)) ⇒ 3
```

```
(string-cursor->index *s* (string-index *s* char-whitespace? 4))
(string-cursor->index *s* (string-index-right *s* char-whitespace?)) ⇒ 8

(string-cursor->index *s* (string-index-right *s* char-whitespace? 0 7)) ⇒ 4

### string-contains

**haystack needle :optional start1 end1 start2 end2**

[Function]

Search for a substring `needle` in a string `haystack`. The substring is searched from left to right (`string-contains`) or right to left (`string-contains-right`). While SRFI-13’s procedures of the same names returns an integer index of the beginning of the substring if found, these procedures returns a string cursor of the beginning of `haystack`. If `haystack` doesn’t contain `needle`, they return `#f`.

### string-for-each-cursor

**proc string :optional start end**

[Function]

Calls `proc` with each cursor of `string`, from left to right, excluding the post-end cursor. This is the cursor version of `string-for-each-index` in SRFI-13 (see Section 11.6.10 [SRFI-13 String mapping], page 699).

```scheme
(define s "abc")
(define (display (string-ref/cursor s c)))
(string-for-each-cursor (^c (display (string-ref/cursor s c))) s)
⇒ displays abc
```

### 11.29 srfi.152 - String library (reduced)

**srfi.152**

This is an improved version of `srfi.13`. The spec is actually smaller than SRFI-13 (hence ‘reduced’ in the title) by removing bells and whistles. The consistency with R7RS and recent srfis are also considered.

The following are built-in. See Section 6.11 [Strings], page 170, for the details.

- `string?`
- `make-string`
- `string`
- `string-length`
- `string->vector`
- `string->list`
- `vector->string`
- `list->string`
- `string-ref`
- `string-set!`
- `substring`
- `string-copy`
- `string=?`
- `string<?`
- `string<=?`
- `string>?`
- `string>=?`
- `string-append`
- `string-join`
- `string-split`
- `string-map`
- `string-for-each`
- `read-string`
- `write-string`
- `string-fill!`

The following procedures are defined to use Unicode string case folding, and `gauche.unicode` module provides them. See Section 9.37.3 [Full string case conversion], page 554, for the details. Note that Gauche’s built-in versions uses character-wise case folding, which differs from string case folding on some characters (German eszett, for example).

- `string-ci=?`
- `string-ci<?`
- `string-ci<=?`
- `string-ci>?`
- `string-ci=>?`

The following procedures are defined in SRFI-13. See Section 11.6 [String library], page 692, for the details. Note: Some of those procedures in SRFI-13 that require predicate allows a char-set or a character to be passed instead (e.g. `string-filter`). In SRFI-152, only predicate is allowed. Our SRFI-152 implementation shares the same procedure with SRFI-13, so they accept the same arguments as SRFI-13’s, but such code won’t be portable as SRFI-152.

- `string-null?`
- `string-any`
- `string-every`
- `string-tabulate`
We describe procedures unique to this module below.

**Function**

```scheme
(string-remove pred string :optional start end)
```

[SRFI-152]{srfi.152}

Returns a substring of `string` between `start` and `end`, except characters that satisfy `pred`. In other words, it is `(string-filter (complement pred) string start end)`.

This is called `string-delete` in srfi.13. Being changed to take only a predicate (but not a character), it is renamed for the consistency of other srfi (e.g. `filter`, `remove` and `delete` in SRFI-1.)

```scheme
(string-remove char-whitespace? "Quick fox jumps over the lazy dog" 3 22)
⇒ "ckfoxjumpsovert"
```

**Function**

```scheme
(string-replicate string from to :optional start end)
```

[SRFI-152]{srfi.152}

Extended substring. It is called `xsubstring` in SRFI-13, but renamed for the consistency.

Extract a substring of `string` between `start` and `end`, and conceptually create a bidirectional infinite string by repeating the substring to both direction. For example, suppose `string` is "abcde", `start` is 1, and `end` is 4. So we repeat the substring "bcd", with one b falling on the index zero:

```
... b c d b c d b c d b ... 
-6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 
```

Then we extract a substring between `from` and `to` out of this infinite string.

```scheme
(string-replicate "abcde" 2 10 1 4) 
⇒ "dbcdbcdb"
```

```scheme
(string-replicate "abcde" -5 -3 1 4) 
⇒ "cdbcdbcdb"
```

**Function**

```scheme
(string-segment string k)
```

[SRFI-152]{srfi.152}

Splits `string` by every `k` characters and returns a list of those strings. The last string may be shorter than `k`.

```scheme
(string-segment "abcdefgijklmn" 3) 
⇒ ("abc" "def" "ghi" "jkl" "mn")
```

We have a similar procedure on lists, `slices` (see Section 6.6.5 [List accessors and modifiers], page 143).

**Function**

```scheme
(string-contains-right string1 string2 :optional start1 end1 start2 end2)
```

[SRFI-152]{srfi.152}

Like `string-contains`, looks for a needle `string2` from a haystack `string1`, but if it is found, returns the start index of the last match, instead of the first match. The returned index is in `string1`. The optional arguments limit the range of a needle and a haystack. If a needle isn’t found, `#f` is returned.
An edge case: If a needle is empty (e.g. string2 is empty, or start2 = end2), it always matches right after the haystack, so end1 is returned.

(string-contains-right "Little Lisper" "Li")
⇒ 7

**string-take-while** string pred :optional start end

**string-take-while-right** string pred :optional start end

[SRFI-152]{srfi.152} Returns the longest prefix or suffix of string in which all characters satisfy pred.

Note: The order of pred and the source object is different from other take-while-style procedures, such as take-while (Section 10.3.1 [R7RS lists], page 593), ideque-take-while (Section 10.3.10 [R7RS immutable deques], page 623), and lseq-take-while (Section 10.3.13 [R7RS lazy sequences], page 633).

**string-drop-while** string pred :optional start end

**string-drop-while-right** string pred :optional start end

[SRFI-152]{srfi.152} Returns the longest prefix or suffix of string in which all characters does not satisfy pred.

Note: The order of pred and the source object is different from other drop-while-style procedures, such as drop-while (Section 10.3.1 [R7RS lists], page 593), ideque-drop-while (Section 10.3.10 [R7RS immutable deques], page 623), and lseq-drop-while (Section 10.3.13 [R7RS lazy sequences], page 633).

**string-span** string pred :optional start end

**string-break** string pred :optional start end

[SRFI-152]{srfi.152} Find the longest prefix of string between start and end in which all characters satisfy / do not satisfy pred, and returns the prefix and the rest of substring as two values.

(string-break "foo@example.com" (cut eqv? <> #\@))
⇒ "foo" and "@example.com"

(string-break "foo@example.com" (cut eqv? <> #\@) 1 10)
⇒ "oo" "@exampl1"

;; This is Gauche specific - a char-set can work as a predicate:
(string-span "VAR_1 = $VAR_2" #\w)
⇒ "VAR_1" and " = $VAR_2"

Note: The order of pred and the source object is different from span and break in scheme.list (see Section 10.3.1 [R7RS lists], page 593).

### 11.30 srfi.154 - First-class dynamic extents

**srfi.154**

This module provides a convenient way to reify the dynamic environment. A continuation captured by call/cc includes the dynamic environment, as well as the control flow. Sometimes you only want the dynamic environment part. A dynamic extent is a reified dynamic environment.

Let’s see an example. You want a procedure that prints out a message to the error port, so you wrote this print-error:

(define print-error
  (lambda (msg) (display msg (current-error-port)))))
However, if `print-error` is called while the current error port is altered, it is affected. It is supposed to be so, that’s the point of `current-error-port`.

\[
\text{(call-with-output-string}
\text{(^p (with-error-to-port p (^[] (print-error "abc\n")))))}
\Rightarrow "abc\n"
\]

If you do want `print-error` to use the error port at the time it is defined, you have to extract the dynamic value at the moment.

\[
\text{(define print-error}
\text{(let1 eport (current-error-port)}
\text{(lambda (msg) (display msg eport)))))}
\]

This would be quickly cumbersome when you need to capture multiple dynamic values, or the original `print-error` is called indirectly and you can’t modify it the way shown above.

Using `dynamic-lambda` addresses this issue. It not only captures the lexical environment, but also the dynamic environment when it is evaluated. So the `current-error-port` in its body returns the current error port at the time of `print-error` being defined, not when it is called:

\[
\text{(define print-error}
\text{(dynamic-lambda (msg) (display msg (current-error-port))))}
\]

`current-dynamic-extent` [Function]

[SRFI-154]{srfi.154} Returns a dynamic-extent object that reifies the current dynamic environment.

`dynamic-extent? obj` [Function]

[SRFI-154]{srfi.154} Returns `#t` iff `obj` is a dynamic-extent object.

`with-dynamic-extent dynext thunk` [Function]

[SRFI-154]{srfi.154} Calls the `thunk` in the dynamic extent `dynext`, and returns the values yielded by `thunk`.

`dynamic-lambda formals body` ... [Macro]

[SRFI-154]{srfi.154} Like `lambda`, but this not only captures the lexical environment, but also the dynamic environment. Yields a procedure.

Note: Since `dynamic-lambda` needs to swap the dynamic environment after executing `body`, the last expression of `body` isn’t called in the tail context even if the procedure created by `dynamic-lambda` is called in tail context.

### 11.31 srfi.160 - Homogeneous numeric vector libraries

`srfi.160` [Module]

This is an enhancement of SRFI-4, Homogeneous vectors (see Section 11.3 [Homogeneous vectors], page 691), and then become a part of R7RS-large (see Section 10.3.3 [R7RS uniform vectors], page 602).

In Gauche, all the procedures in this module are provided as a part of `gauche.uvector` module (see Section 6.13.2 [Uniform vectors], page 198, and described there.

This module only exports the procedures defined in `srfi.160`. 
11.32 srfi.162 - Comparator sublibrary

srfi.162 [Module]
This is a supplement of SRFI-128, comparators. It provides a few comparator procedures, as well as several useful pre-defined comparators, listed below. These pre-defined comparators are already built in Gauche, so see Section 6.2.4.3 [Predefined comparators], page 119, for the details.

- default-comparator
- boolean-comparator
- real-comparator
- char-comparator
- char-ci-comparator
- string-comparator
- string-ci-comparator
- pair-comparator
- list-comparator
- vector-comparator
- eq-comparator
- eqv-comparator
- equal-comparator

 comparator-min comparator obj obj2 ...
 comparator-max comparator obj obj2 ...

[SRFI-162]{srfi.162} Find the object in obj1 obj2 ... that is minimum or maximum compared by comparator. Note: SRFI-114 provides the same procedures.

(comparator-min list-comparator '(a c b) '(a d) '(a c))
⇒ (a c)

 comparator-min-in-list comparator list
 comparator-min-in-list comparator list

[SRFI-162]{srfi.162} Find the object in list that is minimum or maximum compared by comparator. It is an error if list is empty.

11.33 srfi.170 - POSIX API

srfi.170 [Module]
This module provides a portable interface to a part of POSIX API, mainly filesystem interface.

In Gauche, it is a thin wrapper of Gauche’s native system interface (see Section 6.24 [System interface], page 292). You may want to use this module for a portable code.

POSIX Error

posix-error? obj

[SRFI-170]{srfi.170} Returns #t iff obj is a condition thrown due to underlying POSIX-layer error. On Gauche, POSIX-layer condition is the same as <system-error> (see Section 6.19.4 [Conditions], page 253).

posix-error-name err

[SRFI-170]{srfi.170} The argument must be a POSIX-layer condition (<system-error> object). Returns the POSIX error name such as ENOENT.

See also sys-errno->symbol (see Section 6.24.8 [System inquiry], page 313).

posix-error-message err

[SRFI-170]{srfi.170} The argument must be a POSIX-layer condition (<system-error> object). Returns the error message describing the error.

See also sys-strerror (see Section 6.24.8 [System inquiry], page 313).
I/O

open-file fname port-type flags :optional permission buffer-mode [Function]
[SRFI-170]\{srfi.170\} Opens a file named by *fname* and returns a port to read or write the file. You have finer control than open-input-file/open-output-file.

The port-type argument must be either one of the value of the constant binary-input, textual-input, binary-output, textual-output and binary-input/output. The direction of the returned port is determined by this argument.

Gauche doesn’t distinguish textual ports and binary ports.

The flags argument is a bitwise ior of zero or more of the following bitmasks: open/append, open/create, open/exclusive, open/nofollow and open/truncate. These corresponds to the POSIX flags O_APPEND, O_CREAT, O_EXCL, O_NOFOLLOW and O_TRUNC. Gauche exposes the POSIX flags in gauche.fcntl (see Section 9.11 [Low-level file operations], page 432).

Note: POSIX open(2) also requires one of the bitmasks O_RDONLY, O_WRONLY and O_RDWR. These flags are deduced from the port-type argument.

The optional permission is an integer permission bits to be used when a new file is created.

The actual permission bits are also affected by the current umask.

The optional buffer-mode is a value of either one of the followings: buffer-none, buffer-block, and buffer-line.

fd->port fd port-type :optional buffer-mode [Function]
[SRFI-170]\{srfi.170\} Creates a Scheme port wrapping an integer file descriptor *fd*. The descriptor shouldn’t be used by other Scheme ports. After calling this procedure, no I/O should be done via *fd* directly; the descriptor is “owned” by the port, and will be closed once the port is closed.

The port-type argument must be either one of the value of the constant binary-input, textual-input, binary-output, textual-output and binary-input/output. The direction must match how *fd* was opened.

The optional buffer-mode is a value of either one of the followings: buffer-none, buffer-block, and buffer-line.

binary-input [Constant]
textual-input [Constant]
binary-output [Constant]
textual-output [Constant]
binary-input/output [Constant]
[SRFI-170]\{srfi.170\} Constants to be used as port-type argument of open-file and fd->port.

buffer-none [Constant]
buffer-block [Constant]
buffer-line [Constant]
[SRFI-170]\{srfi.170\} Constants to be used as buffer-mode argument of open-file and fd->port.

open/append [Constant]
open/create [Constant]
open/exclusive [Constant]
open/nofollow [Constant]
open/truncate [Constant]
[SRFI-170]\{srfi.170\} Bitmask constants to be used as flags argument of open-file.


File System

create-directory \textit{name} :optional \textit{permission}  
[Function]

create-fifo \textit{name} :optional \textit{permission}  
[Function]

create-hard-link \textit{old-name} \textit{new-name}  
[Function]

create-symlink \textit{old-name} \textit{new-name}  
[Function]

[SRFI-170]\{srfi.170\} Corresponds to Gauche’s built-in \textit{sys-mkdir, sys-mkfifo, sys-link} and \textit{sys-symlink}.

The default value of \textit{permission} is \#o775 for \textit{create-directory}, and \#o664 for \textit{create-fifo} (It’s not optoinal for Gauche’s built-in).

On Windows, \textit{create-fifo} and \textit{create-symlink} are not supported. Procedures exist, but raise an error when called.

read-symlink \textit{name}  
[SRFI-170]\{srfi.170\} Gauche’s \textit{sys-readlink}.

rename-file \textit{old} \textit{new}  
[SRFI-170]\{srfi.170\} Gauche’s \textit{sys-rename}.

delete-directory \textit{name}  
[SRFI-170]\{srfi.170\} Gauche’s \textit{sys-rmdir}. The directory must be empty.

set-file-owner \textit{name} \textit{uid} \textit{gid}  
[Function]

[SRFI-170]\{srfi.170\} Gauche’s \textit{sys-chown}. The \textit{uid} and \textit{gid} arguments should be an integer \textit{uid/gid}, or the value of the constant \textit{owner/unchanged} and \textit{group/unchanged}, respectively, to indicate to keep the original value.

owner/unchanged  
[Constant]

group/unchanged  
[Constant]

[SRFI-170]\{srfi.170\} Constants to be passed as \textit{uid} and \textit{gid} arguments of \textit{set-file-owner}, respectively.

set-file-times \textit{fname} :optional \textit{atime} \textit{mtime}  
[Function]

[SRFI-170]\{srfi.170\} Gauche’s \textit{sys-utime}. The \textit{atime} and \textit{mtime} arguments must be a \textit{<time>} object (see Section 6.24.9 [Time], page 316), or either one of the values of constants \textit{time/now} or \textit{time/unchanged}. The constant \textit{time/now} indicates the time when the procedure is called, and the constant \textit{time/unchanged} indicates the corresponding timestamp remains the same.

time/now  
[Constant]

time/unchanged  
[Constant]

[SRFI-170]\{srfi.170\} Constants that can be passed to \textit{atime} and \textit{mtime} arguments of \textit{set-file-times}.

truncate-file \textit{fname-or-port} \textit{len}  
[Function]

[SRFI-170]\{srfi.170\} The first argument can be a string filename or a open port associated to a file. If it is a filename, \textit{sys-truncate} is called, and if it is a port, \textit{sys-ftruncate} is called.

file-info \textit{fname-or-port} \textit{follow?}  
[Function]

[SRFI-170]\{srfi.170\} Gauche’s \textit{sys-stat, sys-lstat or sys-fstat}.

The first argument can be a string filename or a open port associated to a file. If it is a filename, \textit{sys-stat} or \textit{sys-lstat} is called depending on whether \textit{follow?} is \#t or \#f. If it is a port, \textit{sys-fstat} is called.

Returns a file-info object, which is a \textit{<sys-stat> instance in Gauche (see Section 6.24.4.4 [File stats], page 301)}. 
Chapter 11: Library modules - SRFIs

file-info? obj

[SRFI-170]{srfi.170} Returns #t iff obj is a file-info object, which is a <sys-stat> instance in Gauche (see Section 6.24.4.4 [File stats], page 301).

file-info:device file-info

file-info:inode file-info

file-info:mode file-info

file-info:nlinks file-info

file-info:uid file-info

file-info:gid file-info

file-info:rdev file-info

file-info:size file-info

file-info:blksize file-info

file-info:blocks file-info

file-info:atime file-info

file-info:mtime file-info

file-info:ctime file-info

[SRFI-170]{srfi.170} Accessors of file-info object (<sys-stat>) (see Section 6.24.4.4 [File stats], page 301). Access its dev, ino, mode, nlink, uid, gid, rdev, size, blksize, blocks, atim, mtim, ctim slots, respectively.

Note: <sys-stat>’s atime, mtime and ctime contains integer for the backward compatibility, while file-info:atime, file-info:mtime and file-info:ctime returns <time> object.

file-info:directory? file-info

file-info:fifo? file-info

file-info:symlink? file-info

file-info:regular? file-info

file-info:socket? file-info

file-info:device? file-info

[SRFI-170]{srfi.170} Returns true iff the file-info object (<sys-stat>) is of the respective type. (file-info:device? returns true if the file is either block or character device). It corresponds to the type slot of <sys-stat>.

set-file-mode fname mode-bits

[SRFI-170]{srfi.170} Gauche’s sys-chmod.

directory-files dir :optional dotfiles?

[SRFI-170]{srfi.170} Returns a list of filenames in the directory named dir. Similar to directory-list in file.util (see Section 12.34.1 [Directory utilities], page 881), but (1) it never includes . and .. (as true value is given to :children? argument of directory-list), and (2) it doesn’t include files beginning with . by default, unless dotfiles? is true.

make-directory-files-generator dir :optional dotfiles?

[SRFI-170]{srfi.170} Returns a generator (see Section 9.12 [Generators], page 436) that generates file names in the directory named dir. The names never include . and .., and the filenames beginning with . is omitted unless dotfiles? is true.

open-directory dir :optional dotfiles?

read-directory dir-object

close-directory dir-object

[SRFI-170]{srfi.170} This is a directory scanner API more close to POSIX opendir, readdir and closedir. A directory object is opened with open-directory, and its entries (filenames) can be read one by one using read-directory. If the file names are exhausted, read-directory returns an EOF object. A directory object must be closed with close-directory.
The filenames to be retrieved are the same as `directory-files`; that is, . and .. are never included, and names beginning with . is excluded unless `dotfiles?` is true.

```
real-path path
[Function]
[SRFI-170]{srfi.170} Gauche's `sys-realpath`.

file-space path-or-port
[Function]
[SRFI-170]{srfi.170} Returns the free space of a file system where a file specified by `path-or-port` resides, in number of bytes.

temp-file-prefix
[Parameter]
[SRFI-170]{srfi.170} A parameter holding a path prefix to be used by `create-temp-file` and `call-with-temporary-filename`. In Gauche, the default value is set to `(build-path (temporary-directory) (x->string (sys-getpid)))`. See Section 12.34.5 [Temporary files and directories], page 890, for the description of `temporary-directory`.

create-temp-file :optional prefix
[Function]
[SRFI-170]{srfi.170}

call-with-temporary-filename maker :optional prefix
[Function]
[SRFI-170]{srfi.170}

Process state

Note: For SRFI-170's `current-directory`, we reuse `file.util`'s one. See Section 12.34.1 [Directory utilities], page 881, for the description.

umask
[Function]
[SRFI-170]{srfi.170} Returns the current umask value. Gauche's `sys-umask` without argument.

set-umask! mask
[Function]
[SRFI-170]{srfi.170} Sets `mask`, which must be a nonnegative exact integer, as the current umask. Gauche's `sys-umask`, except that it returns unspecified value.

set-current-directory! dir
[Function]
[SRFI-170]{srfi.170} Sets the process's current working directory to `dir`. Gauche's `sys-chdir`. It can also be done with `current-working-directory` in `file.util` (see Section 12.34.1 [Directory utilities], page 881).

pid
[Function]
[SRFI-170]{srfi.170} Returns the current process id. Gauche's `sys-getpid`.

nice :optional delta
[Function]
[SRFI-170]{srfi.170} Increments the current process's nice value by `delta`, which must be an exact integer. The lower the nice value is, the more chance the process gets run. If omitted, `delta` is assumed to be 1. Gauche's `sys-nice`.

user-uid
[Function]
user-gid
[Function]
user-effective-uid
[Function]
user-effective-gid
[Function]
user-supplementary-gids
[Function]
[SRFI-170]{srfi.170} Gauche's `sys-getuid`, `sys-getgid`, `sys-geteuid`, `sys-getegid`, and `sys-getgroups`, respectively.
User and group database access

user-info uid-or-name

[Function]
[SRFI-170]{srfi.170} Returns a user-info object, which is a <sys-passwd> instance in
Gauche, of a user specified by uid-or-name, which must be an exact integer uid or a string
username. If it is an uid, sys-getpwuid is called, and if it is a username sys-getpwnam is
called.

user-info? obj

[Function]
[SRFI-170]{srfi.170} Returns #t iff obj is a user-info object (<sys-passwd>).

user-info:name uinfo

[Function]
user-info:uid uinfo

[Function]
user-info:gid uinfo

[Function]
user-info:home-dir uinfo

[Function]
user-info:shell uinfo

[Function]
user-info:full-name uinfo

[SRFI-170]{srfi.170} Returns the value of name, uid, gid, dir, shell and gecos slots of a
user-info object (<sys-passwd>) uinfo.

user-info:parsed-full-name uinfo

[Function]
[SRFI-170]{srfi.170}

group-info gid-or-name

[Function]
[SRFI-170]{srfi.170} Returns a group-info object, which is a <sys-group> instance in
Gauche, of a group specified by gid-or-name, which must be an exact integer gid or a string
groupname. If it is an gid, sys-getgrgid is called, and if it is a groupname sys-getgrnam is
called.

group-info? obj

[Function]
[SRFI-170]{srfi.170} Returns #t iff obj is a group-info object (<sys-group>.

group-info:name ginfo

[Function]
group-info:gid ginfo

[Function]
[SRFI-170]{srfi.170} Returns the value of name and gid slot of a group info (<sys-group>).

Time

posix-time

[Function]
[SRFI-170]{srfi.170} Same as current-time (see Section 6.24.9 [Time], page 316). Return
value is a <time> instance of time-utc.

monotonic-time

[Function]
[SRFI-170]{srfi.170} Returns a <time> instance of time-monotonic, representing a elapsed
time since an unspecified epoch. It is guaranteed to monotonically increasing during the
lifetime of the process, even if the system clock is modified.

Environment variables

See Section 11.20 [Accessing environment variables], page 727, for getting environment variables.
This srfi defines the means to modify them.

set-environment-variable! name value

[Function]
[SRFI-170]{srfi.170} Gauche’s sys-setenv with the third argument being #t (overwrite
the existing value).

delete-environment-variable! name

[Function]
[SRFI-170]{srfi.170} Gauche’s sys-unsetenv.
Terminal device control


11.34 srfi.172 - Two safer subsets of R7RS


This module exports a subset of R7RS-small bindings that can be used relatively safely in the quasi-sandbox environment. To realize a true sandbox, the runtime needs more measures such as preventing resource hogging and the data structure that allows loopholes from being passed to those procedures. However, this module can be a good starting point.

The submodule srfi.172.functional further restricts exported bindings to side-effect free ones.

The bindings exported by srfi.172 are as follows:

Chapter 11: Library modules - SRFIs

The module \texttt{srfi.172.functional} exports the above bindings except for:

- \texttt{set!}
- Procedures whose names end with \texttt{!}: \texttt{bytevector-copy!}, \texttt{bytevector-u8-set!}, \texttt{list-set!}, \texttt{read-bytevector!}, \texttt{set-car!}, \texttt{set-cdr!}, \texttt{string-copy!}, \texttt{string-fill!}, \texttt{string-set!}, \texttt{vector-copy!}, \texttt{vector-fill!}, \texttt{vector-set!}
- Procedures whose names contain \texttt{port}: \texttt{call-with-port}, \texttt{close-input-port}, \texttt{close-output-port}, \texttt{close-port}, \texttt{input-port?}, \texttt{output-port?}, \texttt{port?}, \texttt{textual-port?}
- Procedures whose names begin with \texttt{read}: \texttt{read-bytevector}, \texttt{read-bytevector!}, \texttt{read-char}, \texttt{read-error?}, \texttt{read-line}, \texttt{read-string}, \texttt{read-u8}
- Procedures whose names begin with \texttt{peek}: \texttt{peek-char}, \texttt{peek-u8}
- Procedures whose names begin with \texttt{write}: \texttt{write-bytevector}, \texttt{write-char}, \texttt{write-string}, \texttt{write-u8}

\subsection*{11.35 srfi.173 - Hooks (srfi)}

\texttt{srfi.173} [Module]

This module provides hooks, which manages a list of closures to be called.

It is based on Guile’s hooks, which Gauche supports in \texttt{gauche.hook} module (see Section 9.13 [Hooks], page 447). This srfi is a thin layer on top of \texttt{gauche.hook}.

The following procedures are the same as \texttt{gauche.hook}:

\begin{verbatim}
make-hook    hook?    hook->list

hook-add!  hook proc
[SRFI-173]{srfi.173} Add a procedure \texttt{proc} to a hook \texttt{hook}. The procedure must accept the number of arguments that matches the arity of the hook. It is the same as (add-hook! hook proc) of \texttt{gauche.hook}.

hook-delete! hook proc
[SRFI-173]{srfi.173} Delete \texttt{proc} from \texttt{hook}. If \texttt{proc} isn’t in \texttt{hook}, do nothing. It is the same as (delete-hook! hook proc) of \texttt{gauche.hook}.

hook-reset! hook
[SRFI-173]{srfi.173} Remove all the procedures registered in \texttt{hook}. It is the same as (reset-hook! hook) of \texttt{gauche.hook}.

hook-run hook args ...
[SRFI-173]{srfi.173} Apply all the procedures from \texttt{hook} to the \texttt{args}. The order of the procedure isn’t specified in the srfi, though Gauche preserves the order (see run-hook (see Section 9.13 [Hooks], page 447).
\end{verbatim}
list->hook  arity  list  
[SRFI-173]{srfi.173} Creates a new hook with specified arity (which is a non-negative exact integer), which has the procedures in list. All the procedures must accept arity number of arguments.

list->hook!   hook   list  
[SRFI-173]{srfi.173} Replace the list of procedures in hook with procedures in list. All the procedures must accept the number of arguments the same as hook’s arity.

11.36  srfi.174 - POSIX timespecs

srfi.174  [Module]
Note: This srfi is originally intended to be used with SRFI-170 (see Section 11.33 [POSIX API], page 744), but it turned out this was not enough to support the latter–notably, we need SRFI-19 type argument in the constructor.

Consequently, it is likely that this srfi will be superseded later. We deprecate using this srfi; if you need a portable time representation, SRFI-19 is the best choice at this moment. See Section 11.7 [Time data types and procedures], page 701, for the details.

timespec  secs  nsecs  
[SRFI-174]{srfi.174} Creates and returns a timespec, whose second part is secs and nanosecond part is nsecs. The second part must be an exact integer; if it is positive, it’s the number of seconds since Epoch (Jan 1, 1970 UTC), and if it’s negative, it’s before the Epoch. The nanosecond part is a nonnegative integer less than 1e9, and represents the fraction of seconds in nanosecond resolution. Note that if the second part is negative, the nanosecond part counts toward past.

This is the same as (make-time time-utc nsecs secs) of SRFI-19. Note the order of arguments.

timespec?  obj  
[SRFI-174]{srfi.174} Returns #t iff obj is a <time> object. It doesn’t check the type of time.

timespec-seconds  ts  
timespec-nanosecods  ts  
[SRFI-174]{srfi.174} Returns second and nanosecond part of the <time> object. Same as time-second and time-nanosecond of SRFI-19 (note that we use plurals in this srfi).

inexact->timespec  v  
[SRFI-174]{srfi.174} Convert an inexact real number v to a timespec. The integral part of v is the seconds after (when v is positive) or before (when v is negative) the Epoch, and the fractional part is rounded into nanosecond resolution.

timespec->inexact  ts  
[SRFI-174]{srfi.174} Convert a timespec into an inexact real number in seconds. {srfi.174}

timespec=?  ts1  ts2  
[SRFI-174]{srfi.174} Returns #t iff two timespecs are the same. It is the same as time=? in SRFI-19.

timespec<?  ts1  ts2  
[SRFI-174]{srfi.174} Returns #t iff a timespec ts1 is strictly before the timespec ts2. It is the same as time<? in SRFI-19.
Chapter 11: Library modules - SRFIs


timespec-hash \( ts \)  
[Function]  
[SRFI-174]{srfi.174} Compute a hash value of a timespec \( ts \). It is same as passing a <time> object to default-hash, and affected by the current hash salt value (see Section 6.2.3 [Hashing], page 113).

11.37 srfi.175 - ASCII character library

srfi.175  
[Module]  
This module provides a set of character/string utilities that deal with ASCII characters and codepoints. It is particularly useful for small Scheme implementations that only handles, and a portable code that only concerns, ASCII characters.

Since Gauche supports full Unicode (if it is compiled with utf-8 internal encoding, which is the default), you may not need to use this module to write Gauche-specific code. This is mainly for the compatibility.

One of the characteristics of this module is that many procedures can take an integer in place of a character; an integer between 0 and 127 is interpreted as an ASCII codepoint. An integer outside the range is interpreted as a non-ASCII codepoint. Such argument is noted as char-or-code in the description. It is handy while you’re dealing with a binary file which may contain some ASCII text in it.

ascii-codepoint? \( obj \)  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff \( obj \) is an exact integer between 0 and 127, inclusive.

ascii-bytevector? \( obj \)  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff \( obj \) is a bytevector (u8vector) and all of its elements are between 0 and 127, inclusive.

ascii-char? \( obj \)  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff \( obj \) is an ASCII character.

ascii-string? \( obj \)  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff \( obj \) is a string and all of its elements are ASCII characters.

ascii-control? char-or-code  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff char-or-code is in the range of ASCII control characters

\[(\text{ascii-control? } \#\backslash\text{tab}) \Rightarrow \#t\]
\[(\text{ascii-control? } \#\backslash\text{A}) \Rightarrow \#f\]
\[(\text{ascii-control? } 127) \Rightarrow \#t\]
\[(\text{ascii-control? } 255) \Rightarrow \#f\]

Note: The accepted character range corresponds to char-set:ascii-control (see Section 6.10.2 [Predefined character sets], page 167).

ascii-non-control? char-or-code  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff char-or-code is in the range of ASCII but not a control character.

ascii-space-or-tab? char-or-code  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff char-or-code is a tab or a whitespace.

ascii-other-graphic? char-or-code  
[Function]  
[SRFI-175]{srfi.175} Returns \#t iff char-or-code is in the range of ASCII graphic characters but not a letter nor a digit.
Note: The accepted character range corresponds to \((\text{char-set-difference char-set:ascii-graphic char-set:ascii-letter+digit})\) (see Section 6.10.2 [Predefined character sets], page 167).

**ascii-alphanumeric?** \(\text{char-or-code}\)  
[SRFI-175]{srfi.175} Returns \#t iff \(\text{char-or-code}\) is in the ASCII alphabet or digit range.  
Note: The accepted character range corresponds to \(\text{char-set:ascii-letter+digit}\) (see Section 6.10.2 [Predefined character sets], page 167).

**ascii-alphabetic?** \(\text{char-or-code}\)  
[SRFI-175]{srfi.175} Returns \#t iff \(\text{char-or-code}\) is in the ASCII alphabet range.  
Note: The accepted character range corresponds to \(\text{char-set:ascii-letter}\) (see Section 6.10.2 [Predefined character sets], page 167).

**ascii-numeric?** \(\text{char-or-code}\)  
[SRFI-175]{srfi.175} Returns \#t iff \(\text{char-or-code}\) is in the ASCII digit range.  
Note: The accepted character range corresponds to \(\text{char-set:ascii-digit}\) (see Section 6.10.2 [Predefined character sets], page 167).

**ascii-whitespace?** \(\text{char-or-code}\)  
[SRFI-175]{srfi.175} Returns \#t iff \(\text{char-or-code}\) is one of the ASCII whitespaces.  
Note: The accepted character range corresponds to \(\text{char-set:ascii-whitespace}\) (see Section 6.10.2 [Predefined character sets], page 167).

**ascii-upper-case?** \(\text{char-or-code}\)  
**ascii-lower-case?** \(\text{char-or-code}\)  
[SRFI-175]{srfi.175} Returns \#t iff \(\text{char-or-code}\) is one of upper/lower case ASCII alphabets.  
Note: The accepted character range corresponds to \(\text{char-set:ascii-upper-case}/\text{char-set:ascii-lower-case}\) (see Section 6.10.2 [Predefined character sets], page 167).

**ascii-ci=?** \(\text{char-or-code1 char-or-code2}\)  
**ascii-ci<?** \(\text{char-or-code1 char-or-code2}\)  
**ascii-ci<=?** \(\text{char-or-code1 char-or-code2}\)  
**ascii-ci>?** \(\text{char-or-code1 char-or-code2}\)  
**ascii-ci>=?** \(\text{char-or-code1 char-or-code2}\)  
[SRFI-175]{srfi.175} Each argument may be a character or a codepoint. If the argument falls into ASCII upper case letters, it is converted to the corresponding lower case letter codepoint, and then two are compared as codepoints. Two arguments don’t need to be of the same type.

It is allowed to pass non-ascii range; if it is a non-ASCII character, its codepoint of the native encoding is taken (no case-folding is considered), and compared as codepoints. Of course the result are not portable, except that any non-ASCII character has a codepoint greater than any ASCII characters.

**ascii-string-ci=?** \(\text{string1 string2}\)  
**ascii-string-ci<?** \(\text{string1 string2}\)  
**ascii-string-ci<=?** \(\text{string1 string2}\)  
**ascii-string-ci>?** \(\text{string1 string2}\)  
**ascii-string-ci>=?** \(\text{string1 string2}\)  
[SRFI-175]{srfi.175} Compares two strings with ASCII case folding. Argument strings may contain non-ASCII characters; such characters are compared without case folding.
### Function: ascii-upcase char-or-code

(SRFI-175) If the argument represents an ASCII upper/lower case letter, returns the corresponding lower/upper case character or codepoint. The type of the return value is the same as the argument. If the argument is a character or a codepoint other than that, it is returned as is.

### Function: ascii-downcase char-or-code

(SRFI-175) If the argument represents a ASCII upper/lower case letter, returns the corresponding lower/upper case character or codepoint. The type of the return value is the same as the argument. If the argument is a character or a codepoint other than that, it is returned as is.

### Function: ascii-control->graphic char-or-code

(SRFI-175) ASCII control characters (\x00, \x01 .. \x1f, and \x7f) are sometimes noted as ^@, ^A .. ^_, and ^. These procedures converts the control character and corresponding graphical character (the one comes after ^), and vice versa. The type of the return value is the same as the argument. If the argument isn’t an ASCII control character / a graphical character that corresponds to an ASCII control character, #f is returned.

### Function: ascii-graphic->control char-or-code

(SRFI-175) There are four pairs of brackets in ASCII: ( and ), { and }, [ and ], and < and >. If char-or-code is either one of these eight chars or codepoints, returns the corresponding bracket. The type of the return value is the same as the argument. If the argument is a char or a codepoint other than those brackets, #f is returned.

### Function: ascii-mirror-bracket char-or-code

(SRFI-175) Returns a digit character representing the number n. If n is not an exact integer between 0 and 9, inclusive, #f is returned.

### Function: ascii-upper-case-value char-or-code offset limit

(SRFI-175) If char-or-code is n-th ASCII upper/lower case alphabet, and n is less than limit, returns (+ n offset). Otherwise, returns #f.
Chapter 11: Library modules - SRFIs

11.38 srfi.178 - Bitvector library

Gauche supports all SRFI-178 operations by built-in procedures and the module `gauche.bitvector`. The following procedures are built-in. See Section 6.13.3 [Bitvectors], page 203, for the description.

- `bit->integer`  `bit->boolean`
- `bitvector`  `make-bitvector`  `list->bitvector`
- `bitvector?`  `bitvector-length`
- `string->bitvector`  `bitvector->string`
- `bitvector-ref/int`  `bitvector-ref/bool`
- `bitvector-set!`  `bitvector-copy`  `bitvector-copy!`

The following procedures are provided in Section 9.3 [Bitvector utilities], page 374.

- `bitvector-unfold`  `bitvector-unfold-right`  `bitvector-reverse-copy`
- `bitvector-append`  `bitvector-concatenate`  `bitvector-append-subbitvectors`
- `bitvector-empty?`  `bitvector=?`
- `bitvector-take`  `bitvector-take-right`
- `bitvector-drop`  `bitvector-drop-right`
- `bitvector-segment`
- `bitvector-fold/int`  `bitvector-fold/bool`
- `bitvector-fold-right/int`  `bitvector-fold-right/bool`
- `bitvector-map/int`  `bitvector-map/bool`
- `bitvector-map!/int`  `bitvector-map!/bool`
- `bitvector-map->list/int`  `bitvector-map->list/bool`
- `bitvector-for-each/int`  `bitvector-for-each/bool`
- `bitvector-prefix-length`  `bitvector-suffix-length`
- `bitvector-prefix?`  `bitvector-suffix?`
- `bitvector-pad`  `bitvector-pad-right`
- `bitvector-trim`  `bitvector-trim-right`  `bitvector-trim-both`
- `bitvector-swap!`  `bitvector-reverse!`  `bitvector-reverse-copy!`
- `bitvector->list/int`  `bitvector->list/bool`
- `reverse-bitvector->list/int`  `reverse-bitvector->list/bool`
- `reverse-list->bitvector`
- `bitvector->vector/int`  `bitvector->vector/bool`
- `reverse-bitvector->vector/int`  `reverse-bitvector->vector/bool`
- `vector->bitvector`  `reverse-vector->bitvector`
- `bitvector->integer`
- `integer->bitvector`
- `make-bitvector/int-generator`  `make-bitvector/bool-generator`
- `make-bitvector-accumulator`
- `bitvector-not`  `bitvector-not!`
- `bitvector-and`  `bitvector-and!`
- `bitvector-ior`  `bitvector-ior!`
- `bitvector-xor`  `bitvector-xor!`
- `bitvector-eqv`  `bitvector-eqv!`
- `bitvector-nand`  `bitvector-nand!`
- `bitvector-nor`  `bitvector-nor!`
bitvector-andc1 bitvector-andc1!
bitvector-andc2 bitvector-andc2!
bitvector-orc1 bitvector-orc1!
bitvector-orc2 bitvector-orc2!
broadcast-logical-shift
bitvector-count bitvector-count-run
bitvector-if
bitvector-first-bit
bitvector-field-any? bitvector-field-every?
broadcast-field-clear bitvector-field-clear!
broadcast-field-set bitvector-field-set!
broadcast-field-replace bitvector-field-replace!
broadcast-field-replace-same bitvector-field-replace-same!
broadcast-field-rotate
bitvector-field-flip

11.39 srfi.180 - JSON

This srfi defines the means of parsing and constructing JSON.

In Gauche, this module is implemented on top of rfc.json module (see Section 12.49 [JSON parsing and construction], page 937). Notably, the parameter json-nesting-depth-limit is the same as the one in rfc.json.

Gauche’s rfc.json is more flexible in terms of mapping JSON objects to Scheme objects.
On the other hand, srfi.180 provides a streaming parser/generator, which allows the caller to process input as it is read, instead of waiting the entire input to be parsed.

Predicates and parameters

json-error? obj
[SRFI-180]{srfi.180} JSON reader and writer raise a condition that satisfies this predicate when it encounters invalid JSON syntax and/or object, or the input exceeds the limits specified by json-nesting-depth-limit or json-number-of-character-limit parameters.

Since srfi.180 is implemented on top of rfc.json, which raises a condition <json-parse-error> for input and <json-construct-error> for the output, this predicate simply returns #t iff obj is an instance of either class. See Section 12.49 [JSON parsing and construction], page 937, for the detail of these conditions.

json-error-reason err
[SRFI-180]{srfi.180} Returns a string explaining the reason for the error err, if err is a JSON error object (an object that satisfies json-error?).

In Gauche, a JSON error object is an instance of either <json-parse-error> or <json-construct-error>) conditions, and json-error-reason simply returns the content of its message slot.

json-null? obj
[SRFI-180]{srfi.180} Returns #t iff obj is the symbol null.

json-number-of-character-limit
[SRFI-180]{srfi.180} A parameter that holds a real value. If the number of characters of input JSON text exceeds the value while json-generator, json-fold or json-read is processing the input, a JSON error is thrown.
JSON reader

**json-generator**: *optional port-or-generator*  
[Function]  
[SRFI-180]{srfi.180} Streaming parser. The input *port-or-generator* must be an input port, or a char generator.

Each time it is called, it returns one of the following values, parsed from the input.

- **string**: JSON string.
- **real number**: JSON number.
- **#t, #f**: JSON true and false
- **null**: JSON null
- **array-start**: The beginning of an array. What follows is the array’s element, up to the matching **array-end**.
- **array-end**: The ending of an array.
- **object-start**: The beginning of an object. What follows is alternating string keys and JSON values, up to the matching **object-end**.
- **object-end**: The ending of an object.
- **EOF**: After one top-level JSON value is read, the generator returns EOF.

The generator internally tracks the state, and raises `<json-parse-error>` when the input contains invalid JSON text. See also **json-error?** above.

If the input contains more than one top-level JSON value, you have to call **json-generator** after the previous generator is exhausted.

Note that if a top-level JSON value is a number, **true**, **false** or **null**, the parser need to read one character ahead to recognize the value. So the subsequent call of **json-generator** won’t read a character immediately following those values.

Generally, multiple top-level values uses delimiters for each values so it won’t be an issue. See **json-lines-read** and **json-sequence-read** below.

**json-fold**: proc array-start array-end object-start object-end seed  
:optional port-or-generator  
[Function]  
[SRFI-180]{srfi.180} A procedure to translate JSON parsing result to Scheme object on the fly.

The *port-or-generator* argument is either an input port, or a char generator, defaulted to the current input port. It is first passed to **json-generator** to get a generator of parser events.

Then, **json-fold** retrieves value from the generator and take one of the following actions:

- If it generates a string, a number, a boolean or **null**, calls *(proc obj seed)* where *obj* is the generated value and *seed* is the current seed value, and make the result a new seed value.
- If it generates **array-start**, save the current seed, calls *(array-start seed)* and make the result a new seed value.
- If it generates **array-end**, calls *(array-end seed)*, let the result be *arr*, recover the seed value saved at the corresponding **array-start**, and calls *(proc arr recovered-seed)*. Let the result a new seed value.
• If it generates `object-start`, save the current seed, calls `(object-start seed)` and make the result a new seed value.
• If it generates `object-end`, calls `(object-end seed)`, let the result be `obj`, recover the seed value saved at the corresponding `object-start`, and calls `(proc obj recovered-seed)`. Let the result a new seed value.
• If it generates EOF, returns the seed value as the result of `json-fold`.

**json-read :optional port-or-generator**

*SRFI-180*  
**{srfi.180}** Read one JSON value or object from *port-or-generator*, which should be an input port or a char generator. If it is omitted, the current input port is used. JSON strings and numbers are mapped to Scheme strings and numbers. JSON `true` and `false` become `#t` and `#f`. JSON `null` becomes a symbol `null`. JSON arrays become Scheme vectors, and JSON objects become Scheme alist, in which keys are converted to symbols. If the input contains invalid JSON text, a `<json-parse-error>` is thrown.

See also `parse-json` in *rfc.json* (see Section 12.49 [JSON parsing and construction], page 937).

```scheme
(call-with-input-string
 "{"a":1},{"b":true, "c":"foo"},null"
 json-read)
 ⇒ #(((a . 1))
 ((b . #t) (c . "foo")))
 null)
```

**json-lines-read :optional port-or-generator**

*SRFI-180*  
**{srfi.180}** Returns a generator that yields a JSON values at a time, read from the source in JSON Lines format (*http://jsonlines.org/*), which contains multiple toplevel JSON values separated with `\newline`. The input *port-or-generator* should be an input port or a char generator, defaulted to the current input port.

See `json-read` above about the mapping from JSON values to Scheme values.

**json-sequence-read :optional port-or-generator**

*SRFI-180*  
**{srfi.180}** Returns a generator that yields a JSON values at a time, read from the source in JSON Text Sequence format (*RFC7464*, *https://tools.ietf.org/html/rfc7464*). The input *port-or-generator* should be an input port or a char generator, defaulted to the current input port.

JSON Text Sequence can contain multiple JSON values, each one leaded by one or more consecutive U+001E. If it encounters text unparsable as JSON, that segment (until next U+001E) is silently ignored. Returns a list of read JSON values.

See `json-read` above about the mapping from JSON values to Scheme values.

See also `construct-json` in *rfc.json* (see Section 12.49 [JSON parsing and construction], page 937).

**JSON writer**

**json-accumulator port-or-accumulator**

*SRFI-180*  
**{srfi.180}** This is dual to `json-generator`. The *port-or-accumulator* should be an output port or an accumulator that accepts a character or a string. This procedure returns an accumulator that accepts the events such as `json-generator` generates.

**json-write obj :optional port-or-accumulator**

*SRFI-180*  
**{srfi.180}** Write obj as a JSON to *port-or-accumulator*, which must be an output port or an accumulator that accepts a character or a string.
11.40 srfi.181 - Custom ports

srfi.181

This srfi defines a way to implement a port in Scheme. Gauche has native support of such
ports (see Section 9.40 [Virtual ports], page 572), but this srfi is useful for portable code.

The interface is upper compatible to R6RS.

Note that R7RS Scheme distinguishes binary and textual ports, while Gauche ports can
handle both.

11.40.1 Creating custom ports

make-custom-binary-input-port id read! get-pos set-pos! close

make-custom-textual-input-port id read! get-pos set-pos! close

[SRFI-181]{srfi.181} Creates a new binary and textual input port and returns it, respectively.

The id argument is an arbitrary Scheme procedure. SRFI doesn’t specify how it is used. In
Gauche, id will be returned with port-name procedure (see Section 6.21.3 [Common port
operations], page 260).

The read! argument is a procedure to be called as (read! buffer start count). For
make-custom-binary-input-port, buffer is a bytevector (u8vector). For make-custom-
textual-input-port, buffer is either a string or a vector of characters (Gauche’s implement-
tion always use a vector, but portable code should handle both).

It should generate up to count bytes/characters of data and fill buffer beginning from
start, then return the number of bytes/characters generated. It should generate at least
1 byte/character if there’s still data. To indicate the end of the data, it writes no data and
returns 0.

The get-pos argument is a procedure to be called without arguments, and returns an
implementation-dependent object that indicates the current position of the input stream.
The ‘current position’ is where next read! will generate the data. This can be #f if the port
doesn’t provide positions.

For make-custom-binary-input-port, there’s a special rule that if get-pos returns an exact
integer, it should be a byte position in the stream.

The set-pos! argument is a procedure to be called with one argument, a new position. It
should set the source’s position so that next read! starts generating data from there. The
passed position is something that has been returned by get-pos, or, for make-custom-binary-
input-port, an exact integer that indicates the byte offset from the beginning of the input.
This argument can be #f if the port doesn’t support setting positions. The returned value of
set-pos! is ignored.

If the position passed to set-pos! is invalid, an error that satisfy i/o-invalid-position-
error? should be thrown. Portably, it can be done by throwing a condition created by
make-i/o-invalid-position-error (see Section 11.43 [Port positioning], page 773). For
Gauche-specific code, you can throw a condition <io-invalid-position-error>.

The close argument is a procedure to be called without argument, when the custom port is
closed. It can be #f if you don’t need a special cleanup.

make-custom-binary-output-port id write! get-pos set-pos! close

make-custom-textual-output-port id write! get-pos set-pos! close

[SRFI-181]{srfi.181} Creates a new binary and textual out port and returns it, respectively.
The id argument is an arbitrary Scheme object. This SRFI doesn’t specify how it is used. In Gauche, id will be returned with port-name procedure (see Section 6.21.3 [Common port operations], page 260).

The write! argument is a procedure to be called as (write! buffer start count). For make-custom-binary-output-port, buffer is a bytevector. For make-custom-textual-output-port, buffer is either a string or a vector of characters (Gauche’s implementation always use a vector, but portable code should handle both).

The write! procedure writes data in buffer starting from start, upto count items at maximum, to the external sink. It must return the number of actual items written.

The get-pos argument should be a procedure without taking argument, or #f. If it is a procedure, it should return the position of the sink where the next write! writes to. The position can be an arbitrary Scheme object, but for make-custom-binary-output-port, a position represented as an exact integer should correspond to the byte offset in the port.

The set-pos! argument is a procedure to be called with one argument, a new position, or #f. It should set the sink’s position so that next write! starts to write data from there. The passed position is something that has been returned by get-pos, or, for make-custom-binary-output-port, an exact integer that indicates the byte offset from the beginning of the output. This argument can be #f if the port doesn’t support setting positions. The returned value of set-pos! is ignored.

If the position passed to set-pos! is invalid, an error that satisfy i/o-invalid-position-error? should be thrown. Portably, it can be done by throwing a condition created by make-i/o-invalid-position-error (see Section 11.43 [Port positioning], page 773). For Gauche-specific code, you can throw a condition <io-invalid-position-error>.

The close argument is a procedure to be called without argument, when the port is closed. It can be #f if you don’t need a special cleanup.

The flush argument, if provided and not #f, should be a procedure taking no arguments. It is called when the port is requested to flush the data buffered in the sink, if any.

make-custom-binary-input/output-port id read! write! get-pos set-pos! close :optional flush

Creates a bidirectional binary i/o port. Since Gauche doesn’t distinguish binary and textual ports, you can use the returned port for textual i/o as well, but portable code must avoid it.

(The reason textual input/output port is not defined in the SRFI is that it is difficult to define a consistent semantics agnostic to the internal representation of character stream. In Gauche, we immediately convert characters to the octet stream of internal character encoding.)

The arguments, id, read!, write!, get-pos, set-pos!, close and flush are the same as make-custom-binary-input-port and make-custom-binary-output-port.

11.40.2 Transcoded ports

A transcoded port is a portable way to read/write characters in an encodings other than the system’s default one. This API is defined first in R6RS, and adopted in SRFI-181.

In SRFI-181 (and R6RS) world, strings and characters are all an abstract entity without the concept of encodings (internally, you can think them being encoded in the system’s native encoding), and the explicit encodings only matter when you refer to the outside resource, e.g. files or a binary data represented in a bytevector. Therefore, conversions are only defined between binary ports (external world) and textual ports (internal), or a bytevector (external world) and a string (internal).

Here’s some terms:

Codec A codec specifies a character encoding scheme.
**EOL-style**  Specifies (non)conversion of EOL character(s).

**Transcoder**
A transcoder bundles a codec, an eol-style, and error handling mode.

NB: For Gauche-specific code, you can use `gauche.charconv` module (see Section 9.5 [Character code conversion], page 399) that provides more options.

**Transcoders**

**make-transcoder codec eol-style error-handling**  [Function]  
[SRFI-181]{srfi.181} Creates and returns a transcoder with the given parameters. A transcoder is an immutable object.

The `codec` argument must be a codec object, either created by `make-codec` or one of the predefined codecs; see below.

The `eol-style` argument is one of the following symbols to specify End-of-line style:
- `none`  End-of-line character is passed as-is.
- `lf`  Output port converts `\newline` to LF octet. Input port converts any line ending to `\newline`.
- `crlf`  Output port converts `\newline` to CR-LF octet sequence. Input port converts any line ending to `\newline`.

The `error-handling` argument is a symbol to specify the behavior when an encoding or decoding error occurs. It can be one of the following symbols.
- `replace`  Octets that do not consist a valid character encoding are replaced with `\xFFFD` (or `\x??` if the target encoding of an output transcoding port does not contain `\xFFFD`).
- `raise`  Raise an error with a condition satisfying `i/o-encoding-error?`. The erroneous octet are consumed and next I/O on the port continues with the next character.

**native-transcoder**  [Function]  
[SRFI-181]{srfi.181} Returns a singleton of the transcoder representing systems native (internal) codec and eol-style. In Gauche, the native codec is the codec uses Gauche’s native encoding (returned by `gauche-character-encoding`, see Section 6.9 [Characters], page 160), and eol-style is `none`.

**transcoded-port binary-port transcoder**  [Function]  
[SRFI-181]{srfi.181} Creates a transcoded port wrapping `binary-port`, performing the conversion specified by `transcoder`.

If `binary-port` is an input port, it returns an input port, converting the CES specified in `transcoder` to the system’s native encoding.

If `binary-port` is an output port, it returns an output port, converting the system’s native encoding to the CES specified in `transcoder`.

In Gauche, conversion is done by conversion ports. See Section 9.5.3 [Conversion ports], page 402, for the details.

**bytevector->string bytevector transcoder**  [Function]  
[SRFI-181]{srfi.181} Decode the binary data in `bytevector` as the CES specified by `transcoder`, and returns a string in the native encoding.

It is a wrapper of Gauche’s `ces-convert`; see Section 9.5.3 [Conversion ports], page 402.
string->bytevector string transcoder
[SRFI-181]{srfi.181} Encode the string in the CES specified by transcoder, and returns a bytevector.

It is a wrapper of Gauche’s ces-convert-to; see Section 9.5.3 [Conversion ports], page 402.

Codecs

make-codec name
[SRFI-181]{srfi.181} Returns a codec representing a character encoding scheme named by name. A portable code should only use string for name, while Gauche accepts a symbol as well.

If name isn’t recognized as a supported codec name, a condition that satisfies unknown-encoding-error? is thrown.

unknown-encoding-error? obj
[SRFI-181]{srfi.181} If the system sees unknown or unsupported codec, a condition that satisfies this predicate is thrown.

unknown-encoding-error-name obj
[SRFI-181]{srfi.181} The argument must be an unknown encoding error condition that satisfies unknown-encoding-error?. It returns the name that caused the condition to be thrown.

latin-1-codec
utf-8-codec
utf-16-codec
[SRFI-181]{srfi.181} A pre-defined codecs for latin-1 (ISO8859-1), utf-8, and utf-16.

The utf-16 codec recognizes BOM when used for input; if no BOM is found, UTF-16BE is assumed. When used for output, utf-16 always attaches BOM.

EOL style

native-eol-style
[SRFI-181]{srfi.181} Returns the default eol style. In Gauche, it is none.

Transcoding errors

i/o-decoding-error? obj
[SRFI-181]{srfi.181} When an input transcoded port encounters a sequence that’s not valid for the input codec, a condition that satisfies this predicate is thrown.

In Gauche, such condition is <io-decoding-error>.

i/o-encoding-error? obj
[SRFI-181]{srfi.181} When an output transcoded port encounters a character that can’t be encoded in the output codec, and the handling mode is raise, a condition that satisfies this predicate is thrown.

In Gauche, such condition is <io-encoding-error>.

i/o-encoding-error-char i/o-encoding-condition
[SRFI-181]{srfi.181} Retries the character that caused the <io-encoding-error> is thrown.
11.41 srfi.185 - Linear adjustable-length strings

srfi.185 [Module]
This module provides a linear-update version of string-append and string-replace. "Linear update" means the caller won’t access the first string argument, hence the implementation can reuse it to store the result, for the efficiency.

Note that, in Gauche, a string is just a pointer to an immutable string body. Mutation of a string is actually constructing a new string body and swapping the pointer, so it has no performance advantage to immutable versions. And in fact, we implement these without mutating their arguments.

This module also provides macros with the same name as srfi.118 procedures (see Section 11.25 [Simple adjustable-size strings], page 736), which set! the result to its first argument, so they can work as a drop-in replacement to srfi.118.

(string-append-linear! "abc" "def" #\g "hij")
⇒ "abcdefghij"

(string-replace-linear! "abc" "def" #\g "hij")
⇒ "abcdefg hij"

string-append! dst string-or-char ...
string-replace! dst dst-start dst-end src :optional src-start src-end

These macros expand into the following forms:

(set! dst (string-append-linear! dst string-or-char ...))

Hence they can be used for the code that expects dst to contain the result after calling these macros. Gauche supports generalized set!, so dst can be a procedure call with a setter defined.

(define x (list (string-copy "abc")))

(string-append! (car x) #\d #\e #\f)

x ⇒ ("abcdef")

The code that uses SRFI-118 (see Section 11.25 [Simple adjustable-size strings], page 736) is likely to be replaced using these macros.
11.42 srfi.189 - Maybe and Either: optional container types

srfi.189  [Module]
Maybe and Either types are immutable container types that can “wrap” zero or more values. Maybe can be used in the context where the value(s) may be missing (as opposed to “zero values”). Either can be used in the context where the value(s) can be correct ones or erroneous ones.

If you’re familiar with functional languages like Haskell, you may already know them. They are useful when used in the monadic pattern; that is, you can code as if the chain of calculations never fails, yet whenever the calculation fail at one place, the rest of chain is canceled and just the failure is returned.

Maybe is a union of two types, Just and Nothing. Just wraps valid value(s), while Nothing indicates there’s no meaningful values. Either is a union of two types, Right and Left. Right wraps valid value(s), while Left indicates an error, carrying some information as its payload.

11.42.1 Types and predicates

<maybe>  [Class]
<just>  [Class]
<nothing>  [Class]
{srfi.189} Maybe classes. <just> and <nothing> are subclasses of <maybe>. An instance of <just> carries zero or more values (payload), while an instance of <nothing> doesn’t carry information.

The <maybe> class is an abstract class and can’t create an instance of its own. Instances of <just> and <nothing> should be created with constructor procedures just and nothing.

<either>  [Class]
<right>  [Class]
<left>  [Class]
{srfi.189} Either classes. <right> and <left> are subclasses of <either>. An instance of either class carry zero or more values (payload); it is customary to use <right> to propagate the legitimate results of computation, while use <left> to propagate erroneous conditions whose payload describes what’s wrong.

The <either> class is an abstract class and can’t create an instance of its own. Instances of <right> and <left> should be created with constructor procedures right and left.

maybe? obj  [Function]
just? obj  [Function]
nothing? obj  [Function]
{SRFI-189}srfi.189} Type predicates. Returns #t iff obj is a Maybe, a Just, or a Nothing, respectively.

either? obj  [Function]
right? obj  [Function]
left? obj  [Function]
{SRFI-189}srfi.189} Type predicates. Returns #t iff obj is a Either, a Right, or a Left, respectively.

maybe= elt= maybe1 maybe ...  [Function]
{SRFI-189}srfi.189} Equality predicate of Maybes. It returns #t iff all of maybe1 maybe ... are Nothings, or Justs with their respective payload objects are the same using elt=.
either\(=\) elt\(=\) either\(1\) either \ldots

[SRFI-189]{srfi.189} Equality predicate of Eithers. It returns \#t iff all of either\(1\) either \ldots are the same types (all Rights, or all Lefts), with respective payload objects are the same using elt\(=\).

### 11.42.2 Constructors

**just** \(\text{obj} \ldots\)

[SRFI-189]{srfi.189} Returns a Just with \text{obj} \ldots as its payload.

**nothing**

[SRFI-189]{srfi.189} Returns a Nothing.

**right** \(\text{obj} \ldots\)

[SRFI-189]{srfi.189} Returns a Right or a Left respectively, with \text{obj} \ldots as its payload.

**left** \(\text{obj} \ldots\)

[SRFI-189]{srfi.189} Returns a Right or a Left respectively, with \text{obj} \ldots as its payload.

**list->just** \text{objs}

**list->right** \text{objs}

**list->left** \text{objs}

[SRFI-189]{srfi.189} Returns a Just, Right or Left respectively, with \text{objs} as its payload.

**maybe->either** maybe \text{obj} \ldots

[SRFI-189]{srfi.189} The maybe argument must be a Maybe. If it is a Just, then a Right with the same payload is returned. If it is a Nothing, a Left with \text{obj} \ldots as its payload.

**either->maybe** either

[SRFI-189]{srfi.189} The either argument must be an Either. If it is a Right, then a Just with the same payload is returned. If it is a Left, a Nothing is returned.

**either-swap** either

[SRFI-189]{srfi.189} The either argument must be an Either. If it is a Right, a Left with the same payload argument is returned. If it is a Left, a Right with the same payload argument is returned.

### 11.42.3 Accessors

**maybe-ref** maybe failure :optional success

[SRFI-189]{srfi.189} The maybe argument must be a Maybe. If it’s a Nothing, a procedure failure is tail-called with no arguments. If it’s Just, a procedure success is tail-called with its payload object(s). If success is omitted, values is used.

\[
\text{(maybe-ref (just 1 2) (\^[]) (error 'huh?)) +)}
\][72x784]\(\Rightarrow 3\)

**either-ref** either failure :optional success

[SRFI-189]{srfi.189} The either argument must be an Either. If it’s a Left, a procedure failure is called with its payload object(s). If it’s a Right, a procedure success is called with its payload object(s). If success is omitted, values is used.

**maybe-ref/default** maybe default \ldots

[SRFI-189]{srfi.189} The maybe argument must be a Maybe. If it’s a Nothing, default \ldots are returned as multiple values. If it’s a Just, its payload object(s) is/are returned as multiple values.
either-ref/default maybe default ...  
[Function]
[SRFI-189]{srfi.189} The either argument must be an Either. If it’s a Left, default ... are returned as multiple values (the Left’s payload is discarded). If it’s a Right, its payload object(s) is/are returned as multiple values.

maybe-join maybe  
[Function]
[SRFI-189]{srfi.189} If maybe is a Nothing, it is returned. If it’s a Maybe and its only payload is a Maybe, the inner Maybe is returned. Other cases raise an error.

either-join either  
[Function]
[SRFI-189]{srfi.189} If either is a Left, it is returned. If it is a Right and its only payload is an Either, the inner Either is returned. Other cases raise an error.

maybe-bind maybe mproc mproc2 ...  
[Function]
[SRFI-189]{srfi.189} Monadic bind operation. The maybe argument must be a Maybe. If it is a Nothing, it is returned. If it is a Just, its payload object(s) is/are applied to a procedure mproc, which must return a Maybe. If mproc2 ... are given, the same operation is repeated on them.

\[
(maybe-bind m p p2) \\
≡ (maybe-bind (maybe-bind m p) p2)
\]

either-bind either mproc mproc2 ...  
[Function]
[SRFI-189]{srfi.189} Monadic bind operation. The either argument must be an Either. If it is a Left, it is returned as is. If it is a Right, its payload object(s) is/are applied to a procedure mproc, which must return an Either. If mproc2 ... are given, the same operation is repeated on them.

\[
(either-bind e p p2) \\
≡ (either-bind (either-bind e p) p2)
\]

maybe-compose mproc mproc2 ...  
[Function]

either-compose mproc mproc2 ...  
[Function]
[SRFI-189]{srfi.189} Each argument must be a procedure taking zero or more arguments and return a Maybe/an Either. Returns a procedure that accepts zero or more arguments and returns a Maybe/an Either.

When the returned procedure is called, it first calls mproc; if it returns a Nothing/Left, or there’s no more mprocs, the result is returned. If the result is a Just, its payload is applied to the next mproc, and so on.

\[
(maybe-compose p p2 ...) \\
⇒ (maybe-ref m (\[\] (nothing)) \\
(maybe-compose p p2 ...))
\]

11.42.4 Sequence operations
A Maybe and an Either can be a container of zero or one element, and we have several procedures that employ this view. (Note: For this purpose, we treat multiple payload values as a whole, since they are processed in one step—as if we don’t regard multiple procedure arguments and multiple return values as a sequence of individual values.)

maybe-length maybe  
[Function]
either-length either  
[Function]
[SRFI-189]{srfi.189} Returns 0 if maybe/either is a Nothing/Left, and 1 if it is a Just/Right. An error is thrown if the argument isn’t a Maybe/an Either.
maybe-filter pred maybe

`either-filter pred either obj` ...

[SRFI-189]{srfi.189} If `maybe/either` is a Nothing/Left, returns a Nothing/a Left of `obj` ... . If `maybe/either` is a Just/a Right, apply `pred` on its payload value(s). If `pred` returns a true value, `maybe/either` is returned; otherwise, a Nothing/a Left of `obj` ... is returned.

An error is thrown if `maybe/either` isn’t a Maybe/an Either.

maybe-remove pred maybe

either-remove pred either obj ...

[SRFI-189]{srfi.189} Like `maybe-filter/either-filter`, but the meaning of `pred` is reversed.

maybe-sequence mappable cmap :optional aggregator

either-sequence mappable cmap :optional aggregator

[SRFI-189]{srfi.189} This converts a collection of Maybes/Eithers to a Maybe/an Either of a collection. The input collection and the collection in the output can be of different type.

It’s easier to explain using Haskell-ish type signatures, although it’s not precisely specified. Suppose `Container x` is some kind of a collection of type `x`, and `a*` is multiple values of arbitrary types.

\[
\text{Mappable} = \text{Container Maybe a*}
\]
\[
\text{CMap} = ((\text{Maybe a*} \to b) \to \text{Container Maybe a*} \to \text{Container’ b}
\]
\[
\text{Aggregator} = a* \to b
\]

maybe-sequence :: Mappable \to CMap \to Aggregator \to Container’ b

The `cmap` maps `proc` over the input container (`mappable`), and gathers the result into another container. It can be any containers, as long as it matches the `mappable` argument. For example, `mappable` may be a vector of Maybes, and `cmap` can be `vector-map`–in that case, both `Container` and `Container’` are Vector. Or, `mappable` may be a list of Maybes, and `cmap` can be `(cut map-to <string> <> <>), then `Container` is a list and `Container’` is a string.

The types `a*` and `b` is determined by the `aggregator` procedure, whose default value is `list`.

maybe-map proc maybe

either-map proc either

[SRFI-189]{srfi.189} If `maybe/either` is a Nothing/Left, it is returned as is. If it is a Just/Right, its payload value(s) is/are passed to `proc`, and the result is returned.

maybe-for-each proc maybe

either-for-each proc either

[SRFI-189]{srfi.189} If `maybe/either` is a Nothing/Left, these procedures do nothing. Otherwise, `proc` is applied to the argument’s payload values. The result of `proc` is discarded. Returns an unspecified value.

maybe-fold kons knil maybe

either-fold kons knil either

[SRFI-189]{srfi.189} If `maybe/either` is a Nothing/Left, `knil` is returned. Otherwise, `kons` is called with the argument’s payload values, plus `knil`. What `kons` returns becomes the result.

maybe-unfold p f g seed ...

either-unfold p f g seed ...

[SRFI-189]{srfi.189} First, the stop predicate `p` is applied to `seed` ... . If it returns a true value, a Nothing / a Left of `seed` ... is returned. Otherwise, `g` is applied to `seed` ..., which
should return the same number of values as seeds, and passed to \( p \). If \( p \) returns false, it is an error. If \( p \) returns true, \( mapper \) is applied to seed . . ., then the results are wrapped in a Just/Right to be returned.

### 11.42.5 Protocol converters

**maybe->list** \( maybe \)  
[Function]  
[SRFI-189]{srfi.189} If \( maybe \) is a Just, returns a list of its payload values. If it is a Nothing, an empty list is returned.

**list->maybe** \( lis \)  
[Function]  
[SRFI-189]{srfi.189} If \( lis \) is an empty list, a Nothing is returned. Otherwise, a Just that has elements in \( lis \) as payload values is returned.

Note that \((list->maybe (maybe->list x))\) isn’t an identity mapping—if \( x \) is a Just with zero payload values, you’ll get a Nothing.

**either->list** \( either \)  
[Function]  
[SRFI-189]{srfi.189} If \( either \) is a Right, returns a list of its payload values. If it is a Left, returns an empty string.

**list->either** \( lis \ \text{obj} \ldots \)  
[Function]  
[SRFI-189]{srfi.189} If \( lis \) is an empty list, a Left of \( \text{obj} \ldots \) is returned. Otherwise, a Right that has elements in \( lis \) as payload values is returned.

**maybe->truth** \( maybe \)  
[Function]  
[SRFI-189]{srfi.189} If \( maybe \) is a Nothing, \#f is returned. Otherwise, it must be a Just with one value, and its value is returned.

If \( maybe \) is a Just and it doesn’t have exactly one value, an error is thrown. If you want to deal with arbitrary number of payload values, use \( maybe->list-truth \).

**truth->maybe** \( obj \)  
[Function]  
[SRFI-189]{srfi.189} If \( obj \) is \#f, returns a Nothing. Otherwise, returns a Just with \( obj \) as its payload.

Note that \((truth->maybe (maybe->truth x))\) isn’t an identity mapping—if \( x \) is a Just wrapping \#f, you’ll get a Nothing.

**either->truth** \( either \)  
[Function]  
[SRFI-189]{srfi.189} If \( either \) is a Left, \#f is returned. Otherwise, it must be a Right with one value, and its value is returned.

If \( either \) is a Right and it doesn’t have exactly one value, an error is thrown. If you want to deal with arbitrary number of payload values, use \( either->list-truth \).

**truth->either** \( obj \ \text{fail-obj} \ldots \)  
[Function]  
[SRFI-189]{srfi.189} If \( obj \) is \#f, returns a Left with \( \text{fail-obj} \ldots \) is returned. Otherwise, returns a Right with \( obj \) as its payload.

**maybe->list-truth** \( maybe \)  
[Function]  
[SRFI-189]{srfi.189} Like \( maybe->list \), it returns \#f if \( maybe \) is a Nothing. If \( maybe \) is a Just, however, it returns a list of its payload values.

**list-truth->maybe** \( lis-or-false \)  
[Function]  
[SRFI-189]{srfi.189} The argument must be \#f of a list. If it is \#f, a Nothing is returned. If it is a list, a Just with elements of the list is returned.

\((list-truth->maybe (maybe->list-truth x))\) is an identity mapping.
either->list-truth either
[SRFI-189]{srfi.189} Like either->list, it returns #f if either is a Left. If either is a Right, however, it returns a list of its payload values.

list-truth->either lis-or-false fail-objs ...
[SRFI-189]{srfi.189} The list-or-false argument must be #f of a list. If it is #f, a Left with fail-objs ... is returned. If it is a list, a Right with elements of the list is returned.

maybe->generation maybe
[SRFI-189]{srfi.189} If maybe is a Nothing, an EOF object is returned. Otherwise, it must be a Just with one value, and its value is returned. If maybe is a Just and it doesn’t have exactly one value, an error is thrown.

generation->maybe obj
[SRFI-189]{srfi.189} If obj is an EOF value, a Nothing is returned. Otherwise, a Just wrapping obj is returned.

either->generation either
[SRFI-189]{srfi.189} If either is a Left, an EOF object is returned. Otherwise, it must be a Right with one value, and its value is returned. If either is a Right and it doesn’t have exactly one value, an error is thrown.

generation->either obj fail-objs ...
[SRFI-189]{srfi.189} If obj is an EOF value, a Left with fail-objs ... is returned. Otherwise, a Right wrapping obj is returned.

maybe->values maybe
[SRFI-189]{srfi.189} If maybe is a Just, returns its payload as multiple values. If it is a Nothing, returns no values. (Note that a Just with zero values also returns no values.)

values->maybe producer
[SRFI-189]{srfi.189} It first invokes a procedure producer with no values. If it returns zero values, a Nothing is returned; otherwise, a Just with those values are returned.

either->values either
[SRFI-189]{srfi.189} If either is a Right, returns its payload as multiple values. If it is a Left, returns no values. (Note that a Right with zero values also returns no values.)

values->either producer fail-obj ...
[SRFI-189]{srfi.189} It first invokes a procedure producer with no values. If it returns zero values, a Left with fail-obj ... as its payload is returned. If it returns one or more values, a Right with those values are returned.

maybe->two-values maybe
[SRFI-189]{srfi.189} If maybe is a Just with exactly one value, the value and #t is returned. If maybe is a Nothing, two #f is returned. An error is thrown if maybe has a Just with zero or more than two values.

two-values->maybe producer
[SRFI-189]{srfi.189} The inverse of maybe->two-values. A procedure producer is called with no arguments. It must return two values, a possible payload value, and a boolean. If the second value is true, a Just with the first value is returned. If the second value is #f, a Nothing is returned (the first return value is ignored).
exception->either pred thunk

[SRFI-189]{srfi.189} A procedure thunk is called without argument, wrapped by an exception handler. If thunk raises a condition, it is examined by pred. If pred returns true on the condition, the exception is wrapped by a Left and returned. If pred returns #f, the exception is reraised. If no exception is raised, the result(s) of thunk is wrapped by a Right and returned.

11.42.6 Syntactic utilities

maybe-if mtest then else

[SRFI-189]{srfi.189} If the mtest expression yields a Just, evaluates then. If the mtest expression yields a Nothing, evaluates else. If the mtest expression doesn’t produce a Maybe, an error is thrown.

maybe-and maybe ... 

either-and either ....

[SRFI-189]{srfi.189} Evaluates maybe/either from left to right, as far as each yields a Just/Right. If every expression yields a Just/Right, the last one is returned. If it encounters an expression that yields a Nothing/Left, it stops evaluating the rest of expressions and returns the Nothing/Left.

If expressions yield something other than Maybe/Either, an error is thrown.

maybe-or maybe ... 

either-or either ....

[SRFI-189]{srfi.189} Evaluates maybe/either from left to right, as far as each yields a Nothing/Left. If it encounters an expression that yields a Just/Right, it stops evaluating the rest of expressions and returns it.

If expressions yield something other than Maybe/Either, an error is thrown.

maybe-let* (claw ...) body ...

either-let* (claw ...) body ...

[SRFI-189]{srfi.189} This is a Maybe/Either version of and-let*/either-let*.

Each claw can be either one of the following forms:

identifier

The identifier’s value is taken. It must be a Maybe/an Either, or an error is signaled. If it is a Just/Right, evaluation proceeds to the next claw. If it is a Nothing/Left, evaluation stops and the value is returned immediately.

(identifier expression)

The expression is evaluated. It must yield a Maybe/an Either, or an error is signaled. If it is a Just/Right, identifier is bound to its payload, and the rest of claws and body are processed with the scope of identifier. If it is a Nothing/Left, evaluation stops and the value is returned immediately. An error is signaled if a Just/Right doesn’t carry exactly one value.

(expression)

The expression is evaluated. It must yield a Maybe/an Either, or an error is signaled. If it is a Just/Right, evaluation proceeds to the next claw. If it is a Nothing/Left, evaluation stops and the value is returned immediately.

After all claws are processed and none yields a Nothing/Left, body ... are evaluated.

maybe-let*-values (mv-claw ...) body ...

either-let*-values (mv-claw ...) body ...

[SRFI-189]{srfi.189} Multi-value payload version of maybe-let*/either-let*.
Each claw can be either one of the following forms:

**identifier**

The identifier’s value is taken. It must be a Maybe/an Either, or an error is signaled. If it is a Just/Right, evaluation proceeds to the next claw. If it is a Nothing/Left, evaluation stops and the value is returned immediately.

**(formals expression)**

The formals is the same as the formals of the lambda form, that is, a proper or dotted list of identifiers. The expression is evaluated. It must yield a Maybe/an Either, or an error is signaled. If it is a Just/Right, identifiers in the formals are bound with the payload of the Just/Right, and the rest of claws and body are processed with the scope of those identifiers. If it is a Nothing/Left, evaluation stops and the value is returned immediately. An error is signaled if the formals doesn’t match the payload of the Just/Right.

**(expression)**

The expression is evaluated. It must yield a Maybe/an Either, or an error is signaled. If it is a Just/Right, evaluation proceeds to the next claw. If it is a Nothing/Left, evaluation stops and the value is returned immediately.

After all claws are processed and none yields a Nothing/Left, body . . . are evaluated.

**either-guard pred body . . .**

[Macro]

[SRFI-189]{srfi.189} The body . . . is evaluated, and the value(s) it produces are wrapped in a Right and returned. If an exception occurs in body . . ., the thrown condition is passed to a predicate pred. If the condition satisfies the predicate, it is wrapped in a Left and returned. Otherwise, the condition is reraised with raise-continuable.

11.42.7 Trivalent logic

This section describes procedures that deal with trivalent logic—a value can be a false value (Just #f), a true value (Just with anything other than #f), and Nothing.

If any of the arguments is Nothing, the result becomes Nothing (except tri=?).

All the argument must be Maybe type, or an error is signalled.

**tri-not maybe**

[Function]

[SRFI-189]{srfi.189} Returns Just #t if maybe is trivalent-false, Just #f if maybe is trivalent-true, and Nothing if maybe is Nothing.

**try=? maybe . . .**

[Function]

[SRFI-189]{srfi.189} Returns Just #t if arguments are either all trivalent-true or all trivalent-false. Otherwise return Just #f. Note that if any of the argument is Nothing, the result is Just #f (even all arguments are Nothing).

**try-and maybe . . .**

[Function]

[SRFI-189]{srfi.189} Returns Just #t if all arguments are trivalent-true, Just #f if all arguments are Just but at least one of them is Just #f, and Nothing if any of the arguments is Nothing. If there’s no arguments, Just #t is returned.

This is not a shortcut operation like and.

**try-or maybe . . .**

[Function]

[SRFI-189]{srfi.189} Returns Just #f if all arguments are trivalent-false, Just #t if all arguments are Just but at least one of them is trivalent-true, and Nothing if any of the arguments is Nothing. If there’s no arguments, Just #f is returned.

This is not a shortcut operation like or.
try-merge maybe ...

[SRFI-189]{srfi.189} If all arguments are Nothing, Nothing is returned. Otherwise, first Just is returned.

### 11.43 srfi.192 - Port positioning

**srfi.192**

This srfi defines procedures to get and set the current position of the port.

This feature is already supported in the Gauche core, so the following procedure is described in Section 6.21.3 [Common port operations], page 260.

- port-position
- port-has-port-position?
- set-port-position!
- port-has-set-port-position?

**make-i/o-invalid-position-error pos**

[SRFI-192]{srfi.192} This portably creates a condition suitable to be raised from set-port-position! when the given position object can’t be accepted. The pos argument is the offending position object.

In Gauche, such condition is represented by <io-invalid-position-error> class, which is a subclass of <port-error>.

If you raise a condition created with this procedure from the set-position! callback of the custom ports (see Section 11.40 [Custom ports], page 760), Gauche intercepts it and adds the port information to the condition.

**i/o-invalid-position-error? obj**

[SRFI-192]{srfi.192} Returns #t iff obj is an i/o-invalid-position-error condition (or a compound condition that includes it).

In Gauche, it is the same as (condition-has-type? obj <io-invalid-position-error>).

### 11.44 srfi.193 - Command line

**srfi.193**

This srfi clarifying how the command line arguments can be accessed via R7RS command-line, plus a few supporting APIs.

The following procedures are built-in. See Section 6.24.2 [Command-line arguments], page 294, for the details.

- command-line
- script-file

**command-name**

[SRFI-193]{srfi.193} If the first element of command-line is an empty string, returns #f. Otherwise, returns the first element without directory name and obvious extension (.scm, .exe) stripped.

For example, if you run a Scheme script foo.scm as a program, this procedure returns foo. If you compile your script to an executable on Windows as the name /usr/local/bin/foo.exe, this procedure still returns foo. In general, if you are running a Scheme program as some sort of 'command', this procedure returns its name. The exception is when you're running a REPL, in which case this procedure returns #f.

This is useful for diagnostic messages, for example.

**command-args**

[SRFI-193]{srfi.193} Returns the cdr of (command-line).
**script-directory**

[SRFI-193]{srfi.193} Returns the directory part of (script-file), if it has a string path. It always ends with the directory separator. If (script-file) is #f, #f is returned.

This is useful, for example, to find an auxiliary files relative to the script location.

### 11.45 srfi.196 - Range objects

**srfi.196**

This srfi defines range object, an abstract immutable sequence.

Gauche’s data.range module provides superset of srfi.196. This module re-exports the following procedures from it for the portable code. See Section 12.21 [Range], page 845, for the description of the procedures.

- range
- numeric-range
- iota-range
- vector-range
- string-range
- range-append
- range?
- range=?
- range-length
- range-ref
- range-first
- range-last
- range-split-at
- subrange
- range-segment
- range-take
- range-take-right
- range-drop
- range-drop-right
- range-count
- range-any
- range-every
- range-map
- range-map->list
- range-map->vector
- range-for-each
- range-filter-map
- range-filter-map->list
- range-filter
- range-filter->list
- range-remove
- range-remove->list
- range-fold
- range-fold-right
- range-reverse
- range-index
- range-index-right
- range-take-while
- range-take-while-right
- range-drop-while
- range-drop-while-right
- range->list
- range->vector
- range->string
- vector->range
- vector->generator

### 11.46 srfi.197 - Pipeline operators

**srfi.197**

This module provides a set of macros to compose multiple operations. It is similar to Clojure’s “threading macro”.

When you’re passing a result of some procedure to another procedure and so on, sometimes you get a deeply nested expression:

```
(g (f (e (d (c (b (a arg)))))))
```

In Gauche, you can also write the above expression with $ macro (see Section 4.3 [Making procedures], page 47):

```
($ g $ f $ e $ d $ c $ b $ a arg)
```

Deep nesting is avoided, but it’s still right-to-left, and the placement of argument to receive the previous result is limited to the last position.

With chain macro in this module, you can write it from left to right:

```
(chain (a arg) (b _) (c _) (d _) (e _) (f _) (g _))
```

The _ in the second expressions and after indicates the place where previous result is passed. It is conceptually expanded to the following:

```
(let* (((tmp (a arg))
```
Because the placeholder is explicit, you can pass additional arguments:

\[
\begin{align*}
\text{(chain } x \ (y \ a \ _) \ (z \ b))
\equiv \\
\text{(let* } ((\text{tmp } x) \\
\text{\quad (tmp } (y \ a \ \text{tmp}))) \\
\text{\quad (z \ tmp \ b))}
\end{align*}
\]

Or even use multiple values:

\[
\begin{align*}
\text{(chain } \text{mv-expr } (f \ _) \ (g \ _))
\equiv \\
\text{(let*'-values } (((\text{tmp1 tmp2} \ \text{mv-expr}) \\
\text{\quad ((tmp1 tmp2) \ (f \ tmp1 \ tmp2))}) \\
\text{\quad (g \ tmp1 \ tmp2))}
\end{align*}
\]
(let* ([tmp expr]
    [tmp (if (p? x) (f tmp) tmp))
  (if (q? x) (g tmp) tmp))

chain-lambda [placeholder [ellipsis]] step ...
[SRFI-197]{srfi.197}
(chain-lambda step ...)
≡ (lambda args (chain (apply values args) step ...))

nest [placeholder] step ... initial-value
[SRFI-197]{srfi.197} Similar to chain, except the order of steps is right-to-left.
Each step must have exactly one placeholder symbol, for this macro simply nests the steps:
(nest (f a _) (g _ b) (h _) expr)
≡ (f a (g (h expr) b))

11.47 srfi.207 - String-notated bytevectors

bytestring arg ... 
[SRFI-207]{srfi.207} Each argument must be either one of the followings, and the procedure returns a new bytevector which is concatenation of the given arguments.
  • An exact integer, between 0 and 255 inclusive.
  • An ASCII character.
  • A bytevector
  • A string of ASCII characters
An error is signaled if anything other than above is given.
(bytestring "lor" #\r #\x65 #u8(#x6d))
⇒ #u8(108 111 114 114 101 109) ; #u8"lorem"

make-bytestring lis
[SRFI-207]{srfi.207} Returns a new bytevector whose content is a concatenation of elements in lis. Each element must be either an exact integer between 0 and 255 inclusive, an ASCII character, a bytevector or an ASCII string. It is effectively the same as (apply bytestring lis).

make-bytestring! bv at lis
[SRFI-207]{srfi.207} Fill a bytevector bv starting from index at with the elements in lis. Each element must be either an exact integer between 0 and 255 inclusive, an ASCII character, a bytevector or an ASCII string.
Returns an undefined value.
(rlet1 bv (make-u8vector 10)
  (make-bytestring! bv 1 "(ab #\c #u8"de")")
⇒ #u8(0 97 98 99 100 101 0 0 0 0))
bytevector->hex-string bv
hex-string->bytevector string

[SRFI-207]{srfi.207} Convert a bytevector to hexadecimal digit string, and vice versa.

A condition that satisfies `bytestring-error?` is thrown by `hex-string->bytevector` if string contains a letter not valid as a hexadecimal digits.

```scheme
(byvector->hex-string #u8"aloha")
⇒ "616c6f6861"
(hex-string->bytevector "616C6F6861")
⇒ #u8(97 108 111 104 97) ; #u8"aloha"
```

bytevector->base64 bv :optional digits
base64->bytevector string :optional digits

[SRFI-207]{srfi.207} Encode a bytevector `bv` to Base64 encoded string, and vice versa.

A condition that satisfies `bytestring-error?` is thrown by `base64->bytevector` if string contains a letter not valid as a hexadecimal digits. Whitespace characters in string are silently ignored, though.

By default, they use standard Base64 mappings. You can specify a string of two ASCII characters in `digits` to alter mapping of 62nd and 63rd digits; the 'URL-safe' variant of Base64 can be achieved by passing "-_" as `digits`.

See Section 12.42 [Base64 encoding/decoding], page 921, for Gauche-specific Base64 library that provides more APIs.

bytestring->list bv :optional start end

[SRFI-207]{srfi.207} Converts each element of a bytevector `bv` to either a character (if the element is an integer between 32 and 127, inclusive) or an integer, and returns a list of converted elements. If you pass the resulting list to `make-bytestring`, you can obtain the same bytevector as `bv`.

If you pass the optional `start` and `end` indexes, only the specified range of `bv` is considered.

make-bytestring-generator arg . . .

[SRFI-207]{srfi.207} Returns a generator that works as the same as `(uvector->generator (bytestring arg . . .))`, but this can skip creating intermediate bytestrings.

bytestring-pad bv len char-or-u8
bytestring-pad-right bv len char-or-u8

[SRFI-207]{srfi.207} Returns a newly allocated bytevector of at least `len` long. If `bv` is shorter than `len`, the rest is padded with `char-or-u8` (which can be either an ASCII character or an exact integer between 0 and 255). The pad is added to the left with `bytestring-pad`, and to the right with `bytestring-pad-right`.

This is a bytestring version of `string-pad` and `string-pad-right` (see Section 11.6.4 [SRFI-13 String selection], page 694).

bytestring-trim bv pred
bytestring-trim-right bv pred
bytestring-trim-both bv pred

[SRFI-207]{srfi.207} Remove octets that satisfies `pred` at the beginning / at the end / at both ends of a bytevector `bv`. The procedure `pred` receives an exact integer between 0 and 255.

They always return a fresh bytevector, even when no octets are trimmed.

This is a bytestring version of `string-trim`, `string-trim-right`, and `string-trim-both` (see Section 11.6.4 [SRFI-13 String selection], page 694).
bytestring-replace bv1 bv2 start1 end1 :optional start2 end2
[Function]
[SRFI-207]{srfi.207} Returns a new bytevector whose content is a copy of a bytevector bv1, except the part beginning from the index start1 (inclusive) and ending at the index end1 (exclusive) is replaced by a bytevector bv2. When optional start2 and end2 arguments are given, bv2 is trimmed first accordingly. The size of the gap, (~ end1 start1), doesn’t need to be the same as the size of the inserted bytevector. Effectively, this is the same as the following code.

(bytevector-append (subseq bv1 0 start1)
(subseq bv2 start2 end2)
(subseq bv1 end1 (bytevector-length bv1)))

This is a bytestring version of string-replace (see Section 11.6.12 [SRFI-13 Other string operations], page 700).

bytestring<? bv1 bv2
bytestring<=? bv1 bv2
bytestring>? bv1 bv2
bytestring=>? bv1 bv2
[Function]
[SRFI-207]{srfi.207} Compare two bytevectors lexicographically. These are bytestring version of string<? (see Section 6.11.8 [String comparison], page 177).

There’s no bytestring=?, for you can check equality of bytevectors by bytevector=? in scheme.bytevector and gauche.uvector (see Section 9.38.5 [Bytevector compatibility], page 569).

Note that bytevector-comparator compares bytevectors differently. The comparator first compares the lengths, and only uses elements when both vectors are of the same length (see Section 6.2.4.3 [Predefined comparators], page 119).

(bytestring<? #u8"abc" #u8"ac") ⇒ #t
(<? bytevector-comparator #u8"abc" #u8"ac") ⇒ #f)

This contrasts SRFI-207’s view of bytevectors as a string of bytes, with the view of them as an array of bytes.

bytestring-index bv pred :optional start end
bytestring-index-right bv pred :optional start end
[Function]
[SRFI-207]{srfi.207} Returns the first or last index of the octet that satisfies pred in a bytevector bv. The optional start and end arguments limit the range to search.

These are bytestring version of string-index and string-index-right (see Section 11.6.7 [SRFI-13 String searching], page 697).

bytestring-break bv pred
bytestring-span bv pred
[Function]
[SRFI-207]{srfi.207}

bytestring-join bv-list delim :optional grammar
[Function]
[SRFI-207]{srfi.207}

bytestring-split bv delim :optional grammar
[Function]
[SRFI-207]{srfi.207}

read-textual-bytestring prefix :optional port
[Function]
[SRFI-207]{srfi.207}

write-textual-bytestring bv :optional port
[Function]
[SRFI-207]{srfi.207}
write-binary-bitestring  port  arg  ...  
[SRFI-207]{srfi.207}

bytestring-error?  obj  
[SRFI-207]{srfi.207}

11.48  srfi.209 - Enums and enum sets

srfi.209  
 Enums are objects to represent a finite set of options. Traditionally Lisp-family languages have used symbols or keywords for the purpose, but it is error prone that you can pass arbitrary symbols which can only be checked at runtime.

The enums in this srfi allows the user to define and use enums statically, that is, checked at macro-expansion time.

There are three kind of objects involved in the enum system:

Enum type  
An enum type defines all possible enums that can belong to the type. It can be created at macro-expansion time with define-enum, or at run-time with make-enum-type. It is an immutable object.

Enum  
An enum is an object that belong to a certain enum type. Each enum has a name (a symbol unique within the enum type), an ordinal (nonnegative contiguous exact integer), and a value (arbitrary Scheme value, defaults to the name). Enums are instantiated at the same time as an enum type is instantiated. It is also an immutable object.

Enum set  
An enum set is a subset of enums from the enums in a certain enum type. An efficient set operation over an enum set is provided.

Enum types

define-enum  type-name  (name-value  ...)  make-set  
[SRFI-209]{srfi.209}  The high-level declarative interface. This defines a new enum type and interface macros.

The enums the enum type contains are specified with name-value  ..., each of which must be either a symbol, or a form (symbol value). If it is a symbol, it is used both as the enum name and its value. If it is (symbol value), the symbol is used as the enum name and the value is used as its value.

The type-name argument must be an identifier and is bound to a macro, which takes a single symbol as an argument. When invoked, it expands into an enum object.

The make-set argument must also be an identifier and is bound to a macro, which takes zero or more symbols and expands into an enum set object.

Since type-name and make-set are macros, the given enum name(s) can be statically checked at expansion time.

(define-enum fruits (apple orange banana mango papaya) fruit-set)

(fruits apple) ⇒ #<enum apple>  
(fruits mango) ⇒ #<enum mango>

(fruit-set orange mango papaya)  
⇒ #<enum-set fruits (orange mango papaya)>
**make-enum-type**  \texttt{list} :optional \texttt{name}  

A procedural interface to create enum type and associated enums. This procedure first creates enums according to \texttt{list}. Each element must be either a symbol, or a form \texttt{(symbol value)}. If it’s the former, the symbol is used both as the enum name and its value. If it’s the latter, the symbol is used as the enum name and the value is used as its value. Note that enum objects can only be created with this procedure (or the high-level macro) along with an enum type.

Each enum also gets a nonnegative integer ordinal, in the order of \texttt{list}, starting from zero. The optional name argument is a Gauche’s extension. It attaches name to the created enum-type. It is only for debugging aid, and printed when you write the enum type.

Enum type is immutable. Once created, its enum members cannot be changed.

\begin{verbatim}
(make-enum-type '(one two three))
⇒ #<enum-type (one ...)>

(make-enum-type '(one two three) 'counts)
⇒ #<enum-type counts>
\end{verbatim}

**enum-type?**  \texttt{obj}  

Returns \texttt{#t} if \texttt{obj} is an enum-type, \texttt{#f} otherwise.

**enum-type-size**  \texttt{etype}  

Returns the number of enums the enum type \texttt{etype} has.

**enum-min**  \texttt{etype}  

**enum-max**  \texttt{etype}  

Returns the minimum and maximum enum the enum type \texttt{etype} has.

\begin{verbatim}
(define fruits (make-enum-type '(apple orange banana mango papaya)))

(enum-min fruits) ⇒ #<enum apple>

(enum-max fruits) ⇒ #<enum papaya>
\end{verbatim}

They throw an error if \texttt{etype} has no enums.

**enum-type-enums**  \texttt{etype}  

Returns a list of enums the enum type \texttt{etype} has, in the order of their ordinals.

\begin{verbatim}
(define fruits (make-enum-type '(apple orange banana mango papaya)))

(enum-type-enums fruits)
⇒ (#<enum apple> #<enum orange> #<enum banana> #<enum mango> #<enum papaya>)
\end{verbatim}

**enum-type-names**  \texttt{etype}  

Returns a list of names of enums the enum type \texttt{etype} has, in the order of their ordinals.

\begin{verbatim}
(define fruits (make-enum-type '(apple orange banana mango papaya)))

(enum-type-names fruits)
⇒ (apple orange banana mango papaya)
\end{verbatim}

**enum-type-values**  \texttt{etype}  

Returns a list of the values of the enums the enum type \texttt{etype} has, in the order of their ordinals.
enum-type-contains? etype enum

[SRFI-209]{srfi.209} Returns #t iff enum type etype contains enum, #f otherwise.

enum-name->enum etype symbol

[SRFI-209]{srfi.209} Returns an enum object in etype whose name is symbol. If etype doesn’t have an enum of such name, #f is returned.

enum-ordinal->enum etype n

[SRFI-209]{srfi.209} Returns n-th enum of etype. If n is over the range of enums in etype, #f is returned.

enum-name->ordinal etype symbol

[SRFI-209]{srfi.209} Returns the ordinal of the enum in etype whose name is symbol. An error is thrown if etype doesn’t have such symbol.

enum-name->value etype symbol

[SRFI-209]{srfi.209} Returns the value of the enum in etype whose name is symbol. An error is thrown if etype doesn’t have such symbol.

enum-ordinal->name etype n

[SRFI-209]{srfi.209} Returns the name of the n-th enum of etype. An error is thrown if n is out of the range.

enum-ordinal->value etype n

[SRFI-209]{srfi.209} Returns the value of the n-th enum of etype. An error is thrown if n is out of the range.

make-enum-comparator etype

[SRFI-209]{srfi.209} Returns a comparator that can compare two enums that belong to etype. See Section 11.24 [Comparators], page 731, for the details of comparators.

 Enums

enum-type enum

[Function]

enum-name enum

[Function]

enum-ordinal enum

[Function]

enum-value enum

[Function]

[SRFI-209]{srfi.209} Returns enum’s type (<enum-type> instance), enum’s name (symbol), enum’s ordinal (nonnegative exact integer), and enum’s value (any object), respectively.

Note that enum objects can only be constructed with the enum type it belongs (by define-enum or make-enum-type), and they are immutable.

enum? obj

[SRFI-209]{srfi.209} Returns #t if obj is an enum, #f otherwise.

enum=? enum0 enum1 enums ...

[Function]

enum<? enum0 enum1 enums ...

[Function]

enum<=? enum0 enum1 enums ...

[Function]

enum>? enum0 enum1 enums ...

[Function]

enum>=? enum0 enum1 enums ...

[Function]

[SRFI-209]{srfi.209} These compares enums by its ordinals. All the enums must belong to the same enum type, or an error is thrown.

enum-next enum

[SRFI-209]{srfi.209} Returns the next enum of enum, that is, an enum that belongs to the same enum type as enum and has an incremented ordinal. If enum is the last enum of its type, #f is returned.
enum-prev enum  
[SRFI-209]{srfi.209} Returns the previous enum of enum, that is, an enum that belongs to the same enum type as enum and has a decremented ordinal. If enum is the first enum of its type, #f is returned.

Enum sets

enum-type->enum-set etype  
[SRFI-209]{srfi.209} Returns an enum set that contains all enums in the enum type etype.

enum-set etype enum . . .  
[SRFI-209]{srfi.209} Returns an enum set that contains enum . . . of an enum type etype. An error is thrown unless all enum . . . belong to etype.

enum-empty-set etype  
[SRFI-209]{srfi.209} Returns an empty enum set for the enum type etype.

list->enum-set etype enum-list  
[SRFI-209]{srfi.209} Returns an enum set of an enum type etype, containing enums listed in enum-list. If enum-list contains anything other than enums from etype, an error is thrown.

enum-set-projection etype-or-eset eset  
[SRFI-209]{srfi.209} The etype-or-eset argument must be an enum type or an enum set. Returns a new enum set on an enum type etype-or-eset (if it is an enum set, its enum type is used), which contains enums having the same names as the names of enums contained in eset. The eset’s enum type doesn’t need to be the same as the result enum set.

enum-set-copy eset  
[SRFI-209]{srfi.209} Returns a copy of an enum set eset.

enum-set? obj  
[SRFI-209]{srfi.209} Returns #t if obj is an enum set, #f otherwise.

enum-set-contains? eset enum  
[SRFI-209]{srfi.209} Returns #t if enum is a member of eset, #f otherwise.

enum-set-empty? eset  
[SRFI-209]{srfi.209} Returns a #t if eset is empty.

enum-set-disjoint? eset1 eset2  
[SRFI-209]{srfi.209} Returns #t if two enum sets on the same enum type share no common enums. An error is thrown unless eset1 and eset2 are for the same enum type.

enum-set=? eset1 eset2  
[SRFI-209]{srfi.209} This determines subset relationship of two enum sets. Both enum sets must be for the same enum type, or an error is thrown.

enum-set< eset1 eset2  
[SRFI-209]{srfi.209} Returns #t if the names of the member of enum set eset1 is a subset of the names of the member of eset2. Otherwise returns #f. Unlike enum-set<?, two enum set can be of different enum types. Comparison is done with the names only.
enum-set-any? pred eset  
[Function]
SRFI-209}{srfi.209} Returns #t if any or every application of proc to the elements of enum set eset yields true. Otherwise returns #f.

enum-set-every? pred eset  
[Function]
SRFI-209}{srfi.209} Returns #t if any or every application of proc to the elements of enum set eset yields true. Otherwise returns #f.

enum-set-type eset  
[Function]
SRFI-209}{srfi.209} Returns the enum type of the enum set eset.

enum-set-size eset  
[Function]
SRFI-209}{srfi.209} Returns the number of elements in the enum set eset.

enum-set->enum-list eset  
[Function]
SRFI-209}{srfi.209} Returns a list of enums contained in the enum set eset, in the increasing order of enums.

enum-set-adjoin eset enum . . .  
[Function]
SRFI-209}{srfi.209} Returns an enum set that contains all the members in the enum set eset plus enum . . .. All enums must belong to the same enum type of eset.

The linear update version enum-set-adjoin! may modify eset to produce the result.

enum-set-delete eset enum . . .  
[Function]
SRFI-209}{srfi.209} Returns an enum set that contains all the members in the enum set eset except enum . . .. All enums must belong to the same enum type of eset.

The linear update version enum-set-delete! may modify eset to produce the result.

enum-set-delete-all eset list . . .  
[Function]
SRFI-209}{srfi.209} Returns an enum set that contains all the members in the enum set eset the enums in list. An error is thrown if list contains non-enums, or enums that do not belong to the same enum type of eset.

The linear update version enum-set-delete-all! may modify eset to produce the result.

enum-set-count pred eset  
[Function]
SRFI-209}{srfi.209} Applies pred on each enum in eset, and returns the number of times when pred yields a true value.

enum-set-filter pred eset  
[Function]
SRFI-209}{srfi.209} Returns a new enum set that contains the enums that satisfy or do not satisfy pred.

enum-set-map->list proc eset  
[Function]
SRFI-209}{srfi.209} Applies proc on each enum in eset, and returns a list of the results, in the increasing order of enums.

enum-set-for-each proc eset  
[Function]
SRFI-209}{srfi.209} Applies proc on each enum in eset in the increasing order. The results of proc are discarded.

enum-set-fold kons knil eset  
[Function]
SRFI-209}{srfi.209}
Chapter 11: Library modules - SRFIs

### 11.49 srfi.210 - Procedures and syntax for multiple values

**srffi.210**  
This module provides macros and procedures to blend multiple-value expressions naturally into the ordinary single-value expressions.

**with-values**  
_producer_ consumer

[SRFI-210]{srffi.210} A shorthand notation of (call-with-values (λ[] producer) consumer).

**case-receive**  
_producer_ clause ...

[SRFI-210]{srffi.210} A shorthand notation of (with-values producer (case-lambda clause ...)).
apply/mv operator operand ... producer

[Macro]
[SRFI-210]{srfi.210} Similar to apply, but the multiple values yielded from the last argument (producer) is turned into the tail of the argument list; that is, (apply operator operand ... (values->list producer)).

call/mv proc producer ...

[Macro]
[SRFI-210]{srfi.210} Calls proc with the argument produced by the expression producer ... and 'splicing' their result(s) into a single argument list.

(call/mv list (values 1 2 3) (values 4 5 6)) ⇒ (1 2 3 4 5 6)

list/mv element ... producer

vector/mv element ... producer

box/mv element ... producer

[Macro]
[SRFI-210]{srfi.210} Returns a list, a vector, or a box, consisting of element ... and the value(s) produced by producer.

Single-argument list/mv is the same as Gauche’s values->list (see Section 6.15.10 [Multiple values], page 230).

(list/mv 1 2 (values 3 4 5)) ⇒ (1 2 3 4 5)
(vector/mv 1 2 (values 3 4 5)) ⇒ #(1 2 3 4 5)
(box/mv 1 2 (values 3 4 5)) ⇒ #<mv-box[5] 1 2 3 4 5>

value/mv index element ... producer

[Macro]
[SRFI-210]{srfi.210} Returns the same value as (list-ref (list/mv element ... producer) index).

With no element ..., it is the same as Gauche’s values-ref (see Section 6.15.10 [Multiple values], page 230).

coirity producer

[Macro]
[SRFI-210]{srfi.210} Evaluate producer, and returns the number of values it yielded.

bind/mv producer transducer ...

[Macro]
[SRFI-210]{srfi.210} Evaluate producer, and passes its results to the first transducer as arguments. The values returned by the first transducer are passed to the next transducer as arguments, and so on. Returns the results of the last transducer. The call to the last transducer is tail-call.

If no transducers are given, it returns the results of producer.

list-values lis

vector-values vec

box-values box

[Function]
[SRFI-210]{srfi.210} Turn the content of lis, vec or box into multiple values.

Note that box-values is the same as unbox, but provided here for the naming consistency (see Section 6.17 [Boxes], page 238).

value index obj ...

[Function]
[SRFI-210]{srfi.210} Evaluates obj ..., and returns the index-th value.

identity obj ...

[Function]
[SRFI-210]{srfi.210} Synonym of values. Note that this is different from Gauche’s built-in identity, which takes exactly one argument (see Section 6.15.3 [Combinators], page 220).
compose-left\ transducer \ldots\ [Function]\ [SRFI-210]\{srfi.210\} Each transducer must be a procedure. Returns a procedure that applies the first transducer to its arguments, then applies the next transducer to the result(s) of the first one, and so on. Returns the result(s) returned from the last transducer. If no transducers are given, returned procedure is the same as values.

This can be written using Gauche’s built-in compose as follows (see Section 6.15.3 [Combinators], page 220):

\[
\text{(define (compose-left . procs) (apply compose (reverse procs)))}
\]

compose-right\ transducer \ldots\ [Function]\ [SRFI-210]\{srfi.210\} Each transducer must be a procedure. Returns a procedure that applies the last transducer to its arguments, then applies the previous transducer to the result(s) of the last one, and so on. Returns the result(s) returned from the first transducer. If no transducers are given, returned procedure is the same as values.

This is the same as Gauche’s built-in compose (see Section 6.15.3 [Combinators], page 220).

map-values\ proc\ [Function]\ [SRFI-210]\{srfi.210\} Returns a procedure that takes any number of arguments, applies proc on each argument, and returns the results as multiple values. The behavior is undefined if proc doesn’t return a single value.

\[
((\text{map-values (cut + <> 1)}) 1 2 3)
\]

\[
\Rightarrow 2, 3, 4
\]

bind/list\ lis\ transducer\ \ldots\ [Function]\ [SRFI-210]\{srfi.210\} These are bind operation in Monads. The first argument is the source of data; they’re first expanded into values (for bind/list, each element of lis is taken; for bind/box, it is unboxed, and for bind, obj is the singleton value). Those values are passed as arguments to the first transducer, and its result(s) are passed as argument to the second transducer and so on. The value(s) returned by the last transducer becomes the result(s) of those procedures. These can be written using compose-left and procedure application:

\[
\begin{align*}
\text{(bind/list lis proc \ldots)} & \equiv (\text{apply (compose-left proc \ldots) lis}) \\
\text{(bind/box box proc \ldots)} & \equiv (\text{apply/mv (compose-left proc \ldots) (unbox box)}) \\
\text{(bind obj proc \ldots)} & \equiv ((\text{compose-left proc \ldots}) \text{obj})
\end{align*}
\]

11.50 srfi.216 - SICP prerequisites

srfi.216\ [Module]\[SRFI-216]\{srfi.216\} This module provides several utility procedures to write code and implement exercises of “Structure and Interpretation of Computer Programs”.

We provide this module for the portability, i.e. to run existing code that depends on this module. If you’re studying SICP and writing code by yourself using Gauche, we recommend you to install Gauche-compat-sicp extension (https://github.com/shirok/Gauche-compat-sicp), for it takes care of Gauche-specific matters.

true\ [Constant]\[SRFI-216]\{srfi.216\} Constants bound to \#t, \#f, and \(), respectively.
runtime

[Function]
[SRFI-216]{srfi.216} Returns an exact integer monotonically increasing from certain point of time in the past. The absolute value doesn’t matter, but you can use differences of calls to runtime to compare relative elapsed time.

random n

[Function]
[SRFI-216]{srfi.216} The argument n must be a positive real number. Returns a random nonnegative integer less than n.

If n is exact, an exact integer is returned. Otherwise, an inexact integer is returned.

parallel-execute thunk thunk2 ...

[Function]
[SRFI-216]{srfi.216} Run given thunks in parallel using threads. Returns an undefined value after all thunks are completed.

test-and-set! pair

[Function]
[SRFI-216]{srfi.216} The argument must be a mutable pair. If its car is #f, set it to #t and returns #f. Otherwise returns #t without modifying the argument. The operation is performed atomically.

cons-stream a b

[Macro]
[SRFI-216]{srfi.216} Expands into (cons a (delay b)). Note that the car side of stream (a) is eagerly evaluated.

This is not compatible with util.stream’s stream-cons (see Section 12.90.1 [Stream primitives], page 1033). The reason SICP need this as primitive is that SICP does not use macros, so it can’t define its own syntactic abstraction.

the-empty-stream

[Constant]
[SRFI-216]{srfi.216} A singleton instance of the empty stream. This should be used with the following stream-null?. Note that this is not compatible with stream-null of util.stream (see Section 12.90.1 [Stream primitives], page 1033).

stream-null? obj

[Function]
[SRFI-216]{srfi.216} Returns #t if obj is equivalent to the-empty-stream, #f otherwise. Note that this is not the same as stream-null? of util.stream (see Section 12.90.1 [Stream primitives], page 1033).

11.51 srfi.217 - Integer sets

srfi.217

This srfi defines a set whose element is limited to fixnums.

Although general sets are provided by scheme.set (see Section 10.3.5 [R7RS sets], page 606), this module may take advantage of limited-type elements to optimize storage and operations.

It also provides range-based operations, which aren’t available in the general sets.

Constructors

iset element ...

[Function]
[SRFI-217]{srfi.217} Returns a fresh iset contains element .... Each element must be a fixnum.

iset-unfold p f g seed

[Function]
[SRFI-217]{srfi.217} Constructs an iset whose element is computed procedurally.

The p argument is called with a current seed value. If it returns #t, the iteration ends.
The $f$ argument is called with a current seed value, and returns a fixnum to be included into the set.

The $g$ argument is called with a current seed value, and returns the next seed value.

The seed argument gives the initial seed.

\[
\text{(iset->list (iset-unfold (cut > <> 10) (cut * <> 2) (cut + <> 1) 0))}
\Rightarrow (0 2 4 6 8 10 12 14 16 18 20)
\]

\text{make-range-iset start end :optional step} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Creates an iset that contains each fixnum in \text{(numeric-range start end step)}. See Section 12.21 [Range], page 845, for the details of ranges.

\[
\text{(iset->list (make-range-iset 0 5))}
\Rightarrow (0 1 2 3 4)
\]

\text{Predicates}

\text{iset? obj} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns \texttt{#t} iff obj is an iset.

\text{iset-contains? iset element} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns \texttt{#t} iff iset contains element.

\text{iset-empty? iset} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns \texttt{#t} iff iset is empty.

\text{iset-disjoint? iset1 iset2} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Both arguments must be an iset. Returns \texttt{#t} iff no element belongs to both isets.

\text{Accessors}

\text{iset-member iset element default} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns element if it is contained in iset, otherwise default.

\text{iset-min iset} \quad [\text{Function}]

\text{iset-max iset} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns the minimum and the maximum element in iset. If iset is empty, returns \texttt{#f}.

\text{Updaters}

\text{iset-adjoin iset elt1 elt2 . . .} \quad [\text{Function}]

\text{iset-adjoin! iset elt1 elt2 . . .} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns an iset that includes all the elements in iset, plus elt1 elt2 . . . . An error is thrown if elt1 elt2 . . . include other than fixnums.

The linear update version \text{iset-adjoin!} may reuse iset to store the result, but the caller must always use its return value.

\text{iset-delete iset elt1 elt2 . . .} \quad [\text{Function}]

\text{iset-delete! iset elt1 elt2 . . .} \quad [\text{Function}]

[SRFI-217] \text{[srfi.217]} Returns an iset that includes the elements in iset except the ones matching one of elt1 elt2 . . . .

The linear update version \text{iset-delete!} may reuse iset to store the result, but the caller must always use its return value.
iset-delete-all iset elt-list

iset-delete-all! iset elt-list

[SRFI-217]{srFI.217} The elt-list argument must be a list of fixnums. Returns an iset that includes the elements in iset except the ones in elt-list.

The linear update version iset-delete-all! may reuse iset to store the result, but the caller must always use its return value.

iset-delete-min iset

iset-delete-max iset

iset-delete-min! iset

iset-delete-max! iset

[SRFI-217]{srFI.217} Returns two values: The minimum or the maximum element in iset, and a new iset that contains elements in iset except the minimum/maximum element. An error is thrown if iset is empty.

The linear update version iset-delete-min!/iset-delete-max! may reuse iset to store the result, but the caller must always use its return value.

iset-search iset element failure success

iset-search! iset element failure success

[SRFI-217]{srFI.217} Search iset for matching element from lowest to highest value. They return two values, a (possibly updated) iset, and auxiliary value as explained below. The failure and success arguments are both procedures.

If element is found, the success procedure is called with three arguments: the matching element of iset, a procedure update, and a procedure remove. The success procedure can either return or tail call one of either update or remove.

The update procedure can be invoked with two arguments, new-element and obj; if called, new-element replaces element in iset, and iset-search returns the new iset and obj. The remove procedure can be invoked with one argument, obj. It removes the matching element from iset and iset-search returns the new iset and obj. When those procedures are tail-called from success, those values become the return values of iset-search.

If element is not found, the failure procedure is called with two arguments, insert and ignore, both procedures and failure is expected to tail-call one of them.

The insert procedure can be called with one argument, obj. It causes iset-search to return a new iset that contains all the elements from iset plus element, and obj. The ignore procedure can be called with one argument, obj. It causes iset-search to return the original iset and obj.

The linear update version, iset-search!, may reuse iset for the updated iset.

The whole iset

iset-size iset

[SRFI-217]{srFI.217} Returns the number of elements in iset.

iset-find pred iset failure

[SRFI-217]{srFI.217} Returns the smallest element of iset that satisfies pred. If no element satisfies pred, failure is called with no arguments, and its result is returned from iset-find.

iset-count pred iset

[SRFI-217]{srFI.217} Returns the number of elements in iset that satisfy pred.

iset-any? pred iset

[SRFI-217]{srFI.217} Returns #t if any element in iset satisfies pred, #f otherwise.
### Library modules - SRFIs

**Chapter 11: Library modules - SRFIs**

**iset-every? pred iset**

[SRFI-217]{srfi.217} Returns `#t` if every element in `iset` satisfies `pred`, `#f` otherwise.

## Mapping and folding

**iset-map proc iset**

[SRFI-217]{srfi.217} Creates and returns a new iset whose elements consist of the result of `proc` applied to each element of the given `iset`. An error is signaled if `proc` doesn’t return a fixnum.

Note that the size of the result set may be smaller than the input, if `proc` isn’t injective. The order of `proc`’s application is unspecified.

**iset-for-each proc iset**

[SRFI-217]{srfi.217} Applies `proc` to each element in `iset` in increasing order. The result of `proc` is discarded. Returns unspecified result.

**iset-fold kons knil iset**

**iset-fold-right kons knil iset**

[SRFI-217]{srfi.217} Like `fold/fold-right`, apply `kons` on each element in `iset`, increasing or decreasing order, passing the result to the previous `kons`, or `knil` for the first iteration. Returns the last result of `kons`. If `iset` is empty, `kons` is never called and `knil` is returned as is.

\[
\begin{align*}
\text{(iset-fold cons '()) (iset 2 3 5 7 11)} & \Rightarrow (11 7 5 3 2) \\
\text{(iset-fold-right cons '()) (iset 2 3 5 7 11)} & \Rightarrow (2 3 5 7 11)
\end{align*}
\]

**iset-filter pred iset**

**iset-filter! pred iset**

[SRFI-217]{srfi.217} Returns an iset that contains the elements in `iset` that satisfy `pred`. The linear-update version `iset-filter!` may reuse given `iset` to store the result.

**iset-remove pred iset**

**iset-remove! pred iset**

[SRFI-217]{srfi.217} Returns an iset that contains the elements in `iset` that do not satisfy `pred`. The linear-update version `iset-remove!` may reuse given `iset` to store the result.

**iset-partition pred iset**

**iset-partition! pred iset**

[SRFI-217]{srfi.217} Returns two isets, first one consisting of the elements in `iset` that satisfy `pred`, and the second one consisting of the elements that don’t. The linear-update version `iset-partition!` may reuse given `iset` to store one of the results.

## Copying and conversion

**iset-copy iset**

[SRFI-217]{srfi.217} Returns a copy of `iset`.

**iset->list iset**

[SRFI-217]{srfi.217} Returns a list of elements in `iset`, in increasing order.

**list->iset list**

[SRFI-217]{srfi.217} Returns an iset that contains elements in `list`. The elements in `list` must be fixnums. Duplicate elements are consolidated.
list->iset!  iset list
    [SRFI-217]{srfi.217} Returns an iset consists of the elements from iset and list. All the elements must be fixnums. It may reuse the given iset to store the result.

Subsets
iset=?  iset1 iset2 iset3 ...
    [SRFI-217]{srfi.217} Returns #t iff all given isets contains the same elements.
iset<?  iset1 iset2 iset3 ...
iset<=? iset1 iset2 iset3 ...
iset>?  iset1 iset2 iset3 ...
iset=>? iset1 iset2 iset3 ...
    [SRFI-217]{srfi.217} These compares subset relationships between isets. (iset=? iset1 iset2) is #t iff every element of iset1 is contained in iset2, and so on.

Set theory operations
iset-union  iset1 iset2 iset3 ...
iset-union! iset1 iset2 iset3 ...
    [SRFI-217]{srfi.217} Returns an iset that is a union of given isets. Functional version iset-union always returns a fresh iset. Linear update version iset-union! may reuse iset1 to produce the result.
iset-intersection  iset1 iset2 iset3 ...
iset-intersection! iset1 iset2 iset3 ...
    [SRFI-217]{srfi.217} Returns an iset that is an intersection of the given iset. Functional version iset-intersection always returns a fresh iset. Linear update version iset-intersection! may reuse iset1 to produce the result.
iset-difference  iset1 iset2 iset3 ...
iset-difference! iset1 iset2 iset3 ...
    [SRFI-217]{srfi.217} Returns an iset that has elements in iset1 but not in iset2 iset3 ... . Functional version iset-difference always returns a fresh iset. Linear update version iset-difference! may reuse iset1 to produce the result.
iset-xor  iset1 iset2
iset-xor! iset1 iset2
    [SRFI-217]{srfi.217} Returns an iset that has elements, each of which is either in iset1 or iset2 but not in both. Functional version iset-xor always returns a fresh iset. Linear update version iset-xor! may reuse iset1 to produce the result.

Intervals and ranges
iset-open-interval  iset low high
iset-closed-interval  iset low high
iset-open-closed-interval  iset low high
iset-closed-open-interval  iset low high
    [SRFI-217]{srfi.217} Extract elements in iset that are in the range specified by low and high, which must be real numbers. The 'open' and 'close' in the name indicates whether the boundary is included; iset-open-interval doesn’t include both boundary, iset-close-interval includes both boundary, iset-open-closed-interval includes high but not low set-closed-open-interval inclues low but not high.

(iset->list (iset-open-interval (iset 2 3 5 7 11) 2 7))
⇒ (3 5)
(iset->list (iset-closed-interval (iset 2 3 5 7 11) 2 7))
⇒ (2 3 5 7)
(iset->list (iset-open-closed-interval (iset 2 3 5 7 11) 2 7))
⇒ (3 5 7)
(iset->list (iset-closed-open-interval (iset 2 3 5 7 11) 2 7))
⇒ (2 3 5)

iset= iset k
iset< iset k
iset<= iset k
iset> iset k
iset>= iset k

[SRFI-217] Returns a subset of iset whose elements are equal to, less than, less
than or equal to, greater than, and greater than or equal to, k, which must be a real number.

11.52 srfi.219 - Define higher-order lambda

srffi.219

This srffi enhances define form to allow defining function-returning-function compactly. It
works as follows:

(define ((f a b) c d) ...)  
≡ (define (f a b) (lambda (c d) ...))
≡ (define f (lambda (a b) (lambda (c d) ...)))

(define (((f a) b) c) ...)
≡ (define ((f a) b) (lambda (c) ...))
≡ (define f (lambda (a b) (lambda (c) ...)))
≡ (define f (lambda (a) (lambda (b) (lambda (c) ...))))

This feature has been traditionally supported in many implementations, including Gauche. So you can use this feature without using srffi.193. This module is provided for the portable code.

Note: Gauche has two defines: A “vanilla” define that works as defined in R7RS, and an
“extended” define that supports extended lambda arguments. If you import srffi.193, it
imports the former, hence extended lambda arguments are not available. It’s what you need
for the portable code.

11.53 srffi.221 - Generator/accumulator sub-library

srffi.221

This srffi adds several utility procedures to the generator library (see Section 9.12 [Generators],
page 436, for Gauche’s native support of generators; see Section 10.3.12 [R7RS generators],
page 631, for R7RS-large spec).

gcompose-left constructor operation ...  
gcompose-right constructor operation ...

[SRFI-221] The constructor argument is a thunk that returns a generator. Each operation
is a procedure that takes a generator and returns a generator.

These procedures creates and returns a generator that is a composition of given operations
on top of the one constructed by constructor. gcompose-left applies operations left to right
(left-associative), while gcompose-right does right to left (right-associative).

(use gauche.generator)
(use srfi.221)

(generator->list
  (gcompose-left
    (cut make-iota-generator 100)
    (cut gfilter even? <>)
    (cut ggroup <> 5)))
⇒ ((0 2 4 6 8) (10 12 14 16 18) (20 22 24 26 28)
    (30 32 34 36 38) (40 42 44 46 48) (50 52 54 56 58)
    (60 62 64 66 68) (70 72 74 76 78) (80 82 84 86 88)
    (90 92 94 96 98))

;; The same result can be obtained with:
(generator->list
  (gcompose-right
    (cut ggroup <> 5)
    (cut gfilter even? <>)
    (cut make-iota-generator 100)))

accumulate-generated-values accumulator generator
[Function] [SRFI-221]{srfi.221} Calls generator repeatedly, accumulating the values into accumulator, until generator is exhausted. Then it retrieves and returns the accumulated value. See Section 10.3.12 [R7RS generators], page 631, for the details of accumulators.

It is a generalization of generator->list type of converter; you can pass an accumulator that returns the desired type of result.

genumerate gen
[Function] [SRFI-221]{srfi.221} Returns a generator that yields pairs, each of which consists of an exact integer count and the value yielded by gen. The count starts from 0 and incremented.

(generator->list (genumerate (list->generator '(a b c d e))))
⇒ ((0 . a) (1 . b) (2 . c) (3 . d) (4 . e))

gchoice choice-gen source-gen ...
[Function] [SRFI-221]{srfi.221} All arguments are generators. Returns a generator that yields a value from one of source-gens, according to choice-gen.

The choice-gen must be a generator that yields exact integers between 0 and one minus the number of source-gens.

Each time the resulting generator is invoked, choice-gen is called. If it is exhausted, the resulting generator is also exhausted.

Otherwise, the source-gen indexed by the result of choice-gen is called, and if it isn’t exhausted, the value is returned.

If the selected source-gen is exhausted, choice-gen is retried until non-exhausted source-gen yields a value. If all source-gen is exhausted, the resulting generator is also exhausted.

stream->generator stream
[Function] [SRFI-221]{srfi.221} Returns a generator that yields each element of a lazy stream stream. Lazy streams are defined in R7RS-large scheme.stream (see Section 10.3.14 [R7RS stream], page 635), which is a subset of Gauche’s util.stream (see Section 12.90 [Stream library], page 1033).

generator->stream generator
[Function] [SRFI-221]{srfi.221} Returns a lazy stream whose elements consist of the values generated by generator. Lazy streams are defined in R7RS-large scheme.stream (see Section 10.3.14
[R7RS stream], page 635), which is a subset of Gauche’s util.stream (see Section 12.90 [Stream library], page 1033).

See also generator->1seq (see Section 6.18.2 [Lazy sequences], page 241), which returns a lazy sequence defined in R7RS-large scheme.1seg. In Gauche, lazy sequences are integrated to ordinary lists and more lightweight than lazy streams.

11.54 srfi.222 - Compound objects

**srfi.222**

Compound objects are encapsulation of an immutable sequence of objects. Any objects other than a compound object can be included in a compound object; if you try to create a compound object including compound objects, it is ’flattened’—that is, the source’s subobjects are retrieved and each one is put into the new compound object.

This is an abstraction of the compound condition (see Section 6.19.4 [Conditions], page 253), although Gauche’s compound conditions are not built on this.

This module is mainly provided for the compatibility; used with R7RS records (see Section 9.28 [Record types], page 502), you can have a portable multiple inheritance of records.

**make-compound obj ...**

[SRFI-222]{srfi.222} Returns a compound objects having obj ... as its subobjects. Each obj can any Scheme object, but if obj is a compound object itself, it is expanded into its subobjects:

\[
\begin{align*}
& \text{(make-compound 1 2 3)} \\
& \Rightarrow \langle\text{compound-object (1 2 3)}\rangle \\
& \text{(make-compound 0 (make-compound 1 2 3) 4 5)} \\
& \Rightarrow \langle\text{compound-object (0 1 2 3 4 5)}\rangle
\end{align*}
\]

**compound? obj**

[SRFI-222]{srfi.222} Returns #t iff obj is a compound object.

**compound-subobjects obj**

[SRFI-222]{srfi.222} If obj is a compound, returns a list of its subobjects. Otherwise, returns a 1-element list of obj.

That is, it always returns a list of non-compound objects.

**compound-length obj**

[SRFI-222]{srfi.222} If obj is a compound, returns the number of its subobjects. Otherwise, returns 1.

**compound-ref obj k**

[SRFI-222]{srfi.222} Returns k-th subobject of obj if it is a compound. Otherwise, k must be zero and obj itself is returned.

An error is thrown if k is not a nonnegative exact integer or out of range.

**compound-map mapper obj**

[SRFI-222]{srfi.222} Returns a compound object whose subobjects are result of applying mapper on each subobject of obj. If obj is not a compound, it is treated as if a compound with obj as a sole subobject. Note that if mapper returns a compound, its subobjects are spliced in place. Thus the number of subobjects of the result may differ from the number of subobjects of obj.

\[
\text{(compound-map (cut + 1 <>)) (make-compound 1 2 3))}
\]
\[ \Rightarrow \text{#<compound (2 3 4)>} \]
\[ \text{compound-map (cut make-compound <> 'x) (make-compound 1 2 3))} \]
\[ \Rightarrow \text{#<compound (1 x 2 x 3 x)>} \]
\[ \text{compound-map (cut + 1 <>) 3) } \]
\[ \Rightarrow \text{#<compound (4)>} \]

\text{compound-map->list} \ \text{mapper obj} \quad \mid \text{SRFI-222}\{\text{srfi.222}\}

Returns a list of results of applying \text{mapper} on each subobjects of \text{obj}. If \text{obj} is not a compound, \text{mapper} is directly applied on \text{obj}.

Unlike \text{compound-map}, the result of \text{mapper} appears as is in the result list, even when it is a compound object. Hence the length of the result list always matches \text{(compound-length obj)}.

\[ \text{(compound-map->list (cut + 1 <>) (make-compound 1 2 3))} \]
\[ \Rightarrow \text{(#<compound (1 x)> #<compound (2 x)> #<compound (3 x)>)} \]
\[ \text{(compound-map->list (cut + 1 <>) 3) } \]
\[ \Rightarrow \text{#<compound (4)>} \]

\text{compound-filter pred obj} \quad \mid \text{SRFI-222}\{\text{srfi.222}\}

Collect subobjects of \text{obj} that satisfies \text{pred}, and returns a compound that consists of them. If \text{obj} is not a compound, it is treated as if it is a compound with \text{obj} as a sole subobject.

\[ \text{(compound-filter odd? (make-compound 1 2 3))} \]
\[ \Rightarrow \text{#<compound (1 3)>} \]
\[ \text{(compound-filter odd? 1) } \]
\[ \Rightarrow \text{#<compound (1)>} \]
\[ \text{(compound-filter odd? 2) } \]
\[ \Rightarrow \text{#<compound ()>} \]

\text{compound-predicate pred obj} \quad \mid \text{SRFI-222}\{\text{srfi.222}\}

Returns \#t if \text{obj} satisfies \text{pred}, or any of \text{obj}'s subobjects satisfies \text{pred} when \text{obj} is a compound. Returns \#f otherwise.

\text{compound-access pred acc default obj} \quad \mid \text{SRFI-222}\{\text{srfi.222}\}

If \text{obj} satisfies \text{pred}, \text{acc} is applied at \text{obj} and the result is returned. Otherwise, if \text{obj} is a compound, \text{pred} is applied on each subobject, and \text{acc} is applied on the first subobject that satisfies \text{pred}, and the result is returned. If none of the case is satisfied, \text{default} is returned.

\[ \text{(define-record-type <foo> make-foo foo?} \]
\[ \quad \text{(f foo-f))} \]
\[ \text{(define-record-type <bar> make-bar bar?} \]
\[ \quad \text{(b bar-b))} \]
\[ \text{(define c (make-compound (make-foo "foo!")) (make-bar "bar!")) (make-compound (make-foo "foo!")) (make-bar "bar!")) \]
\[ \text{'(ca . cd))} \]
\[ \text{(compound-access foo? foo-f 'none c) } \Rightarrow \text{"foo!"} \]
\[ \text{(compound-access bar? bar-b 'none c) } \Rightarrow \text{"bar!"} \]
\[ \text{(compound-access pair? car 'none c) } \Rightarrow \text{ca} \]
\[ \text{(compound-access char? char->integer 'none c) } \Rightarrow \text{none} \]
11.55 srfi.226 - Control features

srfi.226
This srfi enhances the semantics of Scheme’s fundamental concepts such as continuations and dynamic environments, to support multi-threaded programs and delimited continuations. Some R7RS primitives such as call/cc and dynamic-wind are extended its semantics to be consistent with the enhanced execution model.

The thread API is mostly compatible with SRFI-18, and so as with Gauche’s thread API (see Section 9.35 [Threads], page 530).

Since the enhanced features are tightly coupled with runtime, most APIs in this srfi is implemented as built-ins in Gauche. This section gives you the overview of SRFI-226 APIs and pointers to the documents.

In 0.9.13, Gauche only implements part of SRFI-226. We’ll cover its entirety in future releases. SRFI-226 is an upward compatible extension of R7RS. However, Gauche has also extended R7RS control features in regards of multi-threaded programs, and there are subtle incompatibilites. We’ll move to full SRFI-226 compatibility, but we also provide backward-compatible interface. If you use srfi.226, you’ll get SRFI-226 behavior.

Here’s a list of SRFI-226 APIs and how Gauche supports them.

Continuation prompts

&continuation
make-continuation-violation
continuation-violation?
continuation-violation-prompt-tag
See Section 6.19.4 [Conditions], page 253,
make-continuation-prompt-tag
default-continuation-prompt-tag
continuation-prompt-tag?
call-with-continuation-prompt
abort-current-continuation
See Section 6.15.8 [Continuation prompts], page 227.

Continuations

call-with-non-composable-continuation
call-with-composable-continuation
Not implemented yet.
call-with-current-continuation
call/cc See Section 6.15.7 [Continuations], page 226.
call-in-continuation
call-in Not implemented yet.
call-with-continuation-barrier
Not implemented yet.
continuation-prompt-available?
Not implemented yet.
dynamic-wind
See Section 6.15.7 [Continuations], page 226.
unwind-protect
See Section 6.19.3.1 [High-level exception handling mechanism], page 249.
Shift and reset

shift-at
reset-at Not implemented yet.

shift
reset See Section 9.26 [Partial continuations], page 485. However, current implementation of shift/reset hasn’t integrated to SRFI-226 model yet.

Inspection

continuation?
non-composable-continuation? Not implemented yet.

Continuation marks

with-continuation-mark
with-continuation-marks
call-with-immediate-continuation-mark
continuation-marks
current-continuation-marks
continuation-mark-set?
continuation-mark-set->list
continuation-mark-set->first
See Section 6.15.9 [Continuation marks], page 229.

continuation-mark-set->list*
continuation-mark-set->iterator Not implemented yet.

make-continuation-mark-key
continuation-mark-key? See Section 6.15.9 [Continuation marks], page 229.

Parameter objects

make-parameter
make-thread-parameter
See Section 6.16.1 [Parameters], page 233. Gauche’s built-in make-parameter works in a compatibility mode for now. If you use srfi.226, make-parameter works as specified in SRFI-226.

parameter?
See Section 6.16.1 [Parameters], page 233.

parameterize
See Section 6.16.1 [Parameters], page 233. Gauche’s built-in parameterize works in a compatibility mode for now. If you use srfi.226, make-parameter works as specified in SRFI-226.

current-parameterization
parameterization?
call-with-parameterization
See Section 6.16.1 [Parameters], page 233.

temporarily
See Section 6.16.2 [Dynamic states], page 236.
Chapter 11: Library modules - SRFIs

Fluids

\begin{verbatim}
define-fluid
define-thread-fluid
define-fluidified
fluid-let
fluid-let*
fluid-parameter
Not implemented yet.
\end{verbatim}

Initial continuations

\begin{verbatim}
call-with-initial-continuation
Not implemented yet.
\end{verbatim}

Promises

\begin{verbatim}
delay
promise?
force
make-promise
See Section 6.18.1 [Delay force and lazy], page 239. Gauche’s built-ins are SRFI-226 compatible.
\end{verbatim}

Exceptions

\begin{verbatim}
with-exception-handler
current-exception-handler
exception-handler-stack
See Section 6.19.3.3 [Low-level exception handling mechanism], page 252. Gauche’s built-ins are SRFI-226 compatible.
raise
raise-continuable
See Section 6.19.2 [Signaling exceptions], page 248.
guard
See Section 6.19.3.1 [High-level exception handling mechanism], page 249.
\end{verbatim}

Conditions

\begin{verbatim}
&parameterization
make-parameterization-condition
parameterization-condition?
condition-parameterization
Not implemented yet.
&continuation-marks
make-continuation-marks-condition
continuation-marks-condition?
condition-continuation-marks
Not implemented yet.
&exception-handler-stack
make-exception-handler-stack-condition
exception-handler-stack-condition?
condition-exception-handler-stack
Not implemented yet.
\end{verbatim}
Time objects

```lisp
(time?
current-time
seconds+ time
)
```

See Section 6.24.9 [Time], page 316.

Threads

```lisp
&thread
make-thread-condition
thread-condition?
&uncaught-exception
make-uncaught-exception
uncaught-exception-condition?
uncaught-exception-condition-reason
&thread-already-terminated
make-thread-already-terminated-condition
thread-already-terminated-condition?
&thread-timeout
make-thread-timeout-condition
thread-timeout-condition?
&thread-abandoned-mutex
make-thread-abandoned-mutex-condition
thread-abandoned-mutex-condition?
&concurrent-modification
make-concurrent-modification-violation
concurrent-modification-violation?
```

See Section 9.35.5 [Thread exceptions], page 543.

```lisp
make-thread
thread?
current-thread
thread-start!
thread-yield!
thread-sleep!
thread-terminate!
thread-schedule-terminate!
thread-join!
```

See Section 9.35 [Threads], page 530.

```lisp
make-mutex
mutex?
mutex-state
mutex-lock!
mutex-unlock!
```

See Section 9.35.4.1 [Mutex], page 536.

```lisp
make-condition-variable
condition-variable?
condition-variable-signal!
condition-variable-broadcast!
```

See Section 9.35.4.2 [Condition variable], page 539.
Chapter 11: Library modules - SRFIs

make-thread-local
thread-local?
tiref
tiset!  See Section 9.35.3 [Thread local storage], page 535.
current-interrupt-level
disable-interrupts!
enable-interrupts!
with-interrupts-disabled
with-interrupts-enabled
thread-interrupt!
Not implemented yet.

11.56 srfi.227 - Optional arguments

srfi.227
srfi.227.definitions

This srfi provies macros to specify optional arguments portably. Gauche supports optional arguments in extended formals (see Section 4.3 [Making procedures], page 47), but it is not portable.

The main module, srfi.227, exports macros opt-lambda, opt*-lambda, let-optionals, and let-optionals*. Note that this let-optionals* has different semantics from Gauche’s built-in let-optionals*.

The submodule srfi.227.definitions exports two more macros, define-optionals and define-optionals*.

opt-lambda opt-formals body ...
opt*-lambda opt-formals body ...

[SRFI-227] srfi.227 Evaluates to a procedure, similar to lambda.

The opt-formals is either one of the following form:

(var ... (optvar init) ...)
(var ... (optvar init) ... . restvar)

Var, optvar, and restvar are identifiers. Each init is an expression.

Var ... are required arguments. Optvar ... are optional arguments, and when no corresponding parameter is provided, init is evaluated and used as the value of the argument.

In the first form, giving excessive arguments throws an error; in the second form, excessive arguments are bound to restvar as a list.

The scope of init differs between opt-lambda and opt-lambda*. With opt-lambda, init is evaluated in the scope where opt-lambda is placed. With opt*-lambda, the scope of inits also includes preceding vars and optvars.

(let ((x 1))
  ((opt-lambda (x (+ x 1))) (list x y)) 10)
⇒ (10 2)

(let ((x 1))
  ((opt*-lambda (x (+ x 1))) (list x y)) 10)
⇒ (10 11)

Note: (opt*-lambda (v ... (w init) ...) body ...) is the same as Gauche’s (lambda (v ... :optional (w init) ...) body ...).
let-optionals  expr  opt-formals  body ...  

let-optionals*  expr  opt-formals  body ...  

[SRFI-227]{srfi.227} Syntax sugar to decompose the result of `expr` using `opt-formals`:

\[
\begin{align*}
\text{(let-optionals  expr  opt-formals  body ...)} & \equiv \left(\text{opt-lambda  opt-formals  body ...} \right) \text{expr} \\
\text{(let-optionals*  expr  opt-formals  body ...)} & \equiv \left(\text{opt*-lambda  opt-formals  body ...} \right) \text{expr}
\end{align*}
\]

Note: Gauche has built-in `let-optionals*`, which is different from this srfi. Gauche’s doesn’t allow required parameters. See Section 4.3 [Making procedures], page 47, for the details.

define-optionals  (name .  opt-formals)  body ...  

define-optionals*  (name .  opt-formals)  body ...  

[SRFI-227]{srfi.227.definitions} These two forms are provided in a submodule `srfi.227.definitions`.

\[
\begin{align*}
\text{(define-optionals  (name .  opt-formals)  body ...)} & \equiv \left(\text{define  name  opt-lambda  opt-formals  body ...} \right) \\
\text{(define-optionals*  (name .  opt-formals)  body ...)} & \equiv \left(\text{define  name  opt*-lambda  opt-formals  body ...} \right)
\end{align*}
\]

11.57  srfi.228 - Composing comparators

srfi.228  

This module provides a few utility procedures to compose comparators for typical aggregates. For the details of comparators, see Section 6.2.4 [Basic comparators], page 116, and see Section 11.24 [Comparators], page 731.

make-wrapper-comparator  type-test  unwrap  contents-comparator  

[SRFI-228]{srfi.228} Creates a comparator for objects that wraps other entities. First it tests the given object with `type-test`; then, it calls `unwrap` on it to extract contents, and use `contents-comparator` on the extracted contents.

For example, suppose you want to deal with a string inside a box. You can create a comparator for such structs (`strbox-comparator`) as follows:

\[
\begin{align*}
\text{(define (strbox?  obj) (and (box?  obj) (string? (unbox obj))))} \\
\text{(define strbox-comparator (make-wrapper-comparator strbox? unbox string-comparator))}
\end{align*}
\]

Then you can compare to strboxes with `<?` etc., or make a hashtable or treemap by passing `strbox-comparator` as a comparator.

make-product-comparator  comparator ...  

[SRFI-228]{srfi.228} Creates a comparator for objects of a `common` type of the types dealt with given comparators. Combined with `make-wrapper-comparator`, this can also be used to create a comparator for the product types.

The type predicate of the resulting comparator is satisfied iff the object satisfies every type predicates of `comparators`.

The equality predicate is satisfied iff two objects are equal to each other according to all `comparators`.
The ordering predicate is defined only if all the comparators have ordering predicate. Two objects can be ordered by applying ordering predicate of comparators left to right. If an ordering predicate returns true, the ordering predicate of the product comparator immediately returns true. Otherwise, two objects are checked with the equality predicate of the current comparator. If two objects are not equal, or no more comparators are left, then the ordering predicate of the product comparator returns #f. Otherwise, the next comparator is tried.

The hash function is defined only if all the comparators have hash functions. Each hash function is called, and the results are combined to produce the final hash value.

The following example defines a comparator of a record type based on the comparators of the fields.

```
(define-record-type <member> make-member member?
  (name member-name) ; string
  (age member-age)) ; integer

(define member-comparator
  (make-product-comparator
    (make-wrapper-comparator member? member-name string-comparator)
    (make-wrapper-comparator member? member-age integer-comparator)))
```

`make-sum-comparator` comparator ...

[SRFI-228]{srfi.228} Creates a comparator for objects of a union type of the types dealt with the given comparators.

The type predicate of the created comparator is satisfied if the given object satisfies a type predicate of any one of the given comparators. The leftmost comparator that the object satisfies is called the relevant comparator.

The equality predicate is satisfied if the two objects share the relevant comparator and are equal to each other with regard to the comparator.

The ordering predicate works as follows: First, relevant comparators of two given objects are picked. If the relevant comparator of the first object is to the left of the relevant comparator of the second, the ordering predicate of the sum comparator is satisfied. If the first one is to the right of the second one, the ordering predicate is unsatisfied. If two objects share the relevant comparator, they are ordered by the ordering predicate of the relevant comparator.

The hash function returns the hash value computed by the relevant comparator of the given object.

`comparator-one` [Variable]

[SRFI-228]{srfi.228} This is a comparator that works as a unit element for the product comparator. It accepts any object and all values are equal; that is, its type predicate and its equality predicate always return #t, its ordering predicate always returns #f, and its hash function always returns 0.

`comparator-zero` [Variable]

[SRFI-228]{srfi.228} This is a comparator that works as a unit element for the sum comparator. Its domain is empty; that is, its type predicate always returns #f, and it is an error to invoke its equality predicate, ordering predicate and hash function.

11.58 srfi.229 - Tagged procedures

`srfi.229` [Module]

This srfi allows to attach auxiliary information (tag) to a procedure. A tag can be attached when a procedure is created, and later retrieved. A procedure itself should be considered as an
immutable entity, so you can’t attach or remove a tag to an existing procedure (although you can attach a mutable structure as a tag and mutate it later; much like the closed environment can be mutated).

```
(lambda/tag expr formals body ...)  [Macro]
[SRFI-229]{srfi.229} Evaluates expr, and returns a procedure that is the same as (lambda formals body ...) except that the value of expr is attached as a tag to the resulting procedure. The tag can be retrieved with procedure-tag.
```

```
case-lambda/tag expr clause ...  [Macro]
[SRFI-229]{srfi.229} Each clause is (formals body ...), much like the ordinary case-lambda.
First evaluates expr, and returns a procedure that is the same as (case-lambda clause ...), except that the value of expr is attached as a tag to the resulting procedure. The tag can be retrieved with procedure-tag.
```

```
(procedure/tag? obj)  [Function]
[SRFI-229]{srfi.229} Returns #t iff obj is a procedure and has a tag.
```

```
(procedure-tag proc)  [Function]
[SRFI-229]{srfi.229} Returns a tag of proc, which must be a tagged procedure. If proc is not a procedure or not tagged, an error is signaled.
```

11.59 srfi.232 - Flexible curried procedures

```
srfi.232  [Module]
This srfi provides macros to define a procedure that can accept less than or more than the given formal parameters. If the number of given parameters is less than the required parameters, it becomes partial application; a procedure that will take the remaining parameters is returned. If the number of given parameters is more than the maximum number of parameters, then the excessive parameters are passed to the result of the original procedure, assuming the original procedure returns a procedure.
Currying in the strict sense converts n-ary procedures to nested 1-ary procedures:

\[
\text{lambda}(\text{a b c body}) \\
\Rightarrow \text{lambda}(\text{a})(\text{lambda}(\text{b})(\text{lambda}(\text{c})(\text{body})))
\]

However, with Scheme, you need nested parentheses to give all the parameters to it:

\[
((f 1 2) 3)
\]

The procedure created with this srfi allows more than one parameters to be passed at once. The following calls to f all yield the same result:

\[
(\text{define-curved}(f \text{ a b c body})
\]

\[
(f 1 2 3)
\]
\[
(((f 1) 2) 3)
\]
\[
((f 1) 2 3)
\]
\[
((f 1) 2 3)
\]

Moreover, the procedure can take more parameters than the formals, assuming that the procedure with full parameters return a procedure that takes extra parameters. In other words, the following equivalence holds:

\[
(f 1 2 3 4 5 6) \equiv ((f 1 2 3) 4 5 6)
\]
curried forms body ...  

**[Macro]**

[SRFI-232]{srfi.232} Returns a procedure just like \((\text{lambda forms body ...})\), except that the returned procedure may take less or more parameters than the specified in \(\text{forms}\).

If it is given with less parameters than the required ones, it works as partial application; the result is a procedure that will accept the remaining parameters.

\[
\begin{align*}
\text{(define f (curried (a b c d) (+ a b c d)))} \\
(\text{f 1 2 3 4}) & \Rightarrow 10 \quad ; \text{ordinary application} \\
(\text{f 1 2}) & \Rightarrow \#<\text{procedure}> \quad ; \text{partial application, like (pa$ f 1 2)} \\
((\text{f 1 2}) 3 4) & \Rightarrow 10 \quad ; \text{fulfilling the remaining argument} \\
(((\text{f 1 2}) 3) 4) & \Rightarrow 10 \quad ; \text{partial application can be nested} \\
(((\text{f 1}) 2 3) 4) & \Rightarrow 10 \quad ; \ldots \text{and can be grouped in any way}
\end{align*}
\]

The procedure can accept more parameters than the ones given to \(\text{forms}\), even \(\text{forms}\) does not specify the “rest” parameter. If \(\text{forms}\) takes the rest parameter, the excess arguments are passed to it. If not, the result of the procedure with required parameters are applied over the excess arguments.

\[
\begin{align*}
\text{(define f (curried (a b c) (^[\text{x}] (* x (+ a b c)))))} \\
(\text{f 1 2 3 4}) & \Rightarrow 24 \quad ; \equiv ((\text{f 1 2 3}) 4)
\end{align*}
\]

If no arguments are given to the procedure, it returns the procedure itself:

\[
(((\text{f})) 1 2 3) \equiv (\text{f 1 2 3})
\]

As a special case, \((\text{curried () body})\) and \((\text{curried identifier body})\) are the same as \((\text{lambda () body})\) and \((\text{lambda identifier body})\), respectively.

**define-curried (name forms) body ...**  

**[Macro]**

[SRFI-232]{srfi.232} A shorthand of \((\text{define name (curried forms body ...))})\).

### 11.60 srfi.235 - Combinators (SRFI)

**srfi.235**  

**[Module]**

This srfi provides combinator utilities. Gauche provides some frequently-used combinators built-in. This module augments it.

The following procedures are built-in; see Section 6.15.3 [Combinators], page 220, for the details.

\[
\text{constantly complement flip swap boolean}
\]

#### Combinator utilities

Procedures that takes procedure(s) and returns a procedure.

**on-left proc**  

**[Function]**

[SRFI-235]{srfi.235} Returns a two argument procedure, which ignores its right argument and calls \(\text{proc}\) with the left argument.

\[
((\text{on-left car}) '(a b) '(c d)) \Rightarrow a
\]

**on-right proc**  

**[Function]**

[SRFI-235]{srfi.235} Returns a two argument procedure, which ignores its left argument and calls \(\text{proc}\) with the right argument.

\[
((\text{on-right car}) '(a b) '(c d)) \Rightarrow c
\]
**Chapter 11: Library modules - SRFIs**

**conjoin** *pred* . . .

[SRFI-235]{srfi.235} Returns a procedure taking any number of arguments and that behaves as follows: For each argument, calls *pred* . . . left to right; if any of *pred* returns *#f*, stop and immediately returns *#f*. If all predicates returns a true value for all arguments, returns the last result. If no *preds* are given, or no arguments are given, returns *#t*.

\[
\begin{align*}
((\text{conjoin char? digit->integer}) \ #\1 \ #\2 \ #\3) & \Rightarrow 3 \\
((\text{conjoin char? digit->integer}) \ #\1 \ #\a \ #\3) & \Rightarrow \#f
\end{align*}
\]

**disjoin** *pred* . . .

[SRFI-235]{srfi.235} Returns a procedure taking any number of arguments and that behaves as follows: For each argument, calls *pred* . . . left to right; if any of *pred* returns a true value, stop and immediately returns that value. If all predicates returns *#f* for all arguments, returns *#f*. If no *preds* are given, or no arguments are given, returns *#f*.

\[
\begin{align*}
((\text{disjoin integer? char?}) \ 'a \ 'b \ 'c) & \Rightarrow \#f \\
((\text{disjoin integer? char?}) \ #\a \ 'b \ 'c) & \Rightarrow \#t
\end{align*}
\]

**each-of** *proc* . . .

[SRFI-235]{srfi.235} Returns a procedure that works, when applied to arguments, applies each *proc* on the arguments in turn. The result of *procs* are discarded.

\[
\begin{align*}
&\quad (\text{define (f x y) (print (+ x y)))} \\
&\quad (\text{define (g x y) (print (* x y)))} \\
&\quad ((\text{each-of f g}) \ 2 \ 7) \\
&\Rightarrow \text{#<undef>}; \text{prints 9 and 14}
\end{align*}
\]

**all-of** *pred*

[SRFI-235]{srfi.235} Returns a procedure that takes one argument, a list. The returned procedure tries to apply *pred* to each element of *list* in turn. If *pred* returns *#f*, immediately returns *#f*. Otherwise, returns the last result of *pred*. If the returned procedure is applied to an empty list, *#t* is returned.

The following equivalence holds:

\[
((\text{all-of pred}) \ \text{list}) \equiv (\text{every pred list})
\]

**any-of** *pred*

[SRFI-235]{srfi.235} Returns a procedure that takes one argument, a list. The returned procedure applies *pred* to each element of *list* in turn. If *pred* returns *#f*, immediately returns *#f*. Otherwise, returns *#t*. If the returned procedure is applied to an empty list, *#f* is returned.

The following equivalence holds:

\[
((\text{any-of pred}) \ \text{list}) \equiv (\text{any pred list})
\]

**on** *reducer mapper*

[SRFI-235]{srfi.235} Returns a procedure such that \((\text{apply (on reducer mapper) args})\) works as if \((\text{apply reducer (map mapper args)})\).

\[
(\text{define square-sum (on + square)})
\]

\[
(\text{square-sum 1 2 3 4 5}) \Rightarrow 55
\]

**left-section** *proc* *arg* . . .

[SRFI-235]{srfi.235} Returns a procedure such that, when applied to more args, calls *proc* with *arg* . . . and the given args. In other words, returns a specialized procedure with the left argument(s) are given as *arg* . . .

In Gauche, this is the same as \((\text{pa$ arg$ . . .})\) (see Section 6.15.3 [Combinators], page 220).
right-section proc arg . . .
  [SRFI-235]{srfi.235}
apply-chain proc proc2 . . .
  [SRFI-235]{srfi.235}
arguments-drop proc n
arguments-drop-right proc n
arguments-take proc n
  [SRFI-235]{srfi.235}
group-by key-proc :optional equal
  [SRFI-235]{srfi.235}

Syntax-like procedures

begin-procedure thunk . . .
  [SRFI-235]{srfi.235} Calls thunk . . . from left to right. Discats their results except the last one(s), which will be the result(s) of this procedure.
  If no thunks are given, returns #<undef>.

if-procedure value then-thunk else-thunk
  [SRFI-235]{srfi.235} If value is true, calls then-thunk and returns its result(s). Otherwise, calls else-thunk and returns its result(s).

when-procedure value thunk . . .
unless-procedure value thunk . . .
  [SRFI-235]{srfi.235}

value-procedure value then-proc else-thunk
  [SRFI-235]{srfi.235}

case-procedure value then-alist :optional else-thunk
  [SRFI-235]{srfi.235}

and-procedure thunk . . .
  [SRFI-235]{srfi.235} Calls thunk . . . left to right, until it yields #f, and returns #f. Otherwise, returns the result of the last thunk. If no thunks are given, returns #t.

eager-and-procedure thunk . . .
  [SRFI-235]{srfi.235} Calls all thunks, then checks their results. If any of the result is #f, returns #f. Otherwise, returns the last result of thunk. If no thunks are given, returns #t.

or-procedure thunk . . .
  [SRFI-235]{srfi.235} Calls thunk . . . left to right, until it yields a true value, and returns that value. If all thunks yields true value, returns #t. If no thunks are given, returns #f.

eager-or-procedure thunk . . .
  [SRFI-235]{srfi.235} Calls all thunks, then checks their results from left to right, and returns the first true value. If all the results are #f, or no thunks are given, returns #f.

funcall-procedure thunk
  [SRFI-235]{srfi.235} Just call thunk and returns its result(s). This is useful to pass to other procedures, e.g. (map funcall-procedure list-of-procedures).
Chapter 11: Library modules - SRFIs

loop-procedure thunk

[Function]
[SRFI-235]{srfi.235} Calls thunk repeatedly forever. Nonlocal escape is the only way to break out.

while-procedure thunk
until-procedure thunk

[SRFI-235]{srfi.235} Calls thunk repeatedly while/until it returns a true value, respectively.

Auxiliary utilities

always obj...

[SRFI-235]{srfi.235} Ignores arguments and always returns #t. Same as (constantly #t).

never obj...

[SRFI-235]{srfi.235} Ignores arguments and always returns #f. Same as (constantly #f).

Note: boolean is provided in the core (see Section 6.4 [Booleans], page 139).

11.61 srfi.239 - Destructuring lists

srfi.239

[Module]
This module provides a pattern-matching macro specialized for the common pattern appearing frequently in list processing code. It can be seen as a subset of match (see Section 12.87 [Pattern matching], page 1024); if you’re writing Gauche-specific code, you don’t need this module. However, this is useful for portable code.

list-case expr clause...

[Macro]
[SRFI-239]{srfi.239} Each clause must be one of the following forms:

- ((var1 . var2) body ...)  
- (() body ...)  
- (var body ...)

Here, var, var1, var2 are identifiers. The order of clauses does not matter, and not all three forms need to be present.

First expr is evaluated. If it is a pair, the first type of clause is selected, and body ... are evaluated with the environment where the pair’s car is bound to var1 and the pair’s cdr is bound to var2. If it is an empty list, the second type of clause is selected, and its body ... are evaluated. Otherwise, the third type of clause is selected, its body ... are evaluated where the value of expr is bound to var.

The identifier _ has special meaning in var, var1, and var2; it is not bound and the corresponding value is simply discarded.

If not all three forms are present and the result of expr does not match the provided clauses, an error is thrown. Note that, unlike match, if the first type of clause isn’t provided, a pair does not match the third type of clause.
12 Library modules - Utilities

12.1 binary.io - Binary I/O

This module provides basic procedures to perform binary I/O of numeric data. Each datum can be read from or written to a port, and got from or put to a uniform vector (see Section 6.13.2 [Uniform vectors], page 198). For structured binary data I/O, more convenient pack utility is implemented on top of this module (see Section 12.2 [Packing binary data], page 811). You might want to use this module directly if you need speed or want a flexible control of endianness.

See also Section 6.13.2 [Uniform vectors], page 198, which provides binary block I/O.

Endianness

Most procedures of this module take an optional endian argument, specifying the byte order of the binary input. It must be either one of symbols big-endian, little-endian, or arm-little-endian. If the endian argument is omitted, the current value of the builtin parameter default-endian is used (see Section 6.3.7 [Endianness], page 138). (For 8-bit I/O procedures like read-u8 the endian argument has no effect, but is accepted for consistency).

I/O using port

read-u8 :optional port endian [Function]
read-u16 :optional port endian [Function]
read-u32 :optional port endian [Function]
read-u64 :optional port endian [Function]

{binary.io} Reads 8, 16, 32 or 64 bit unsigned integer from port with specified endian, respectively. If port is omitted, current input port is used. If port reaches EOF before a complete integer is read, EOF is returned.

read-s8 :optional port endian [Function]
read-s16 :optional port endian [Function]
read-s32 :optional port endian [Function]
read-s64 :optional port endian [Function]

{binary.io} Reads 8, 16, 32 or 64 bit 2’s complement signed integer from port with specified endian, respectively. If port is omitted, current input port is used. If port reaches EOF before a complete integer is read, EOF is returned.

read-uint size :optional port endian [Function]
read-sint size :optional port endian [Function]

{binary.io} More flexible version. Reads size-octet unsigned or signed integer from port with specified endian. If port reaches EOF before a complete integer is read, EOF is returned.

read-ber-integer :optional port [Function]

{binary.io} Reads BER compressed integer a la X.209. A BER compressed integer is an unsigned integer in base 128, most significant digit first, where the high bit is set on all but the final (least significant) byte.

write-u8 val :optional port endian [Function]
write-u16 val :optional port endian [Function]
write-u32 val :optional port endian [Function]
write-u64 val :optional port endian

\{binary.io\} Writes a nonnegative integer val as 8, 16, 32 or 64 bit unsigned integer to port with specified endian, respectively. Val must be within the range of integers representable by the specified bits. When port is omitted, current output port is used.

write-s8 val :optional port endian
write-s16 val :optional port endian
write-s32 val :optional port endian
write-s64 val :optional port endian

\{binary.io\} Writes an integer val as 8, 16, 32 or 64 bit as 2’s complement signed integer to port with specified endian, respectively. Val must be within the range of integers representable by the specified bits. When port is omitted, current output port is used.

write-uint size val :optional port endian
write-sint size val :optional port endian

\{binary.io\} More flexible version. Writes an integer val as unsigned or signed integer of size bytes to port with specified endian. When port is omitted, current output port is used.

write-ber-integer val :optional port

\{binary.io\} Writes a nonnegative integer val in BER compressed integer to port. See read-ber-integer above for BER format.

read-f16 :optional port endian
read-f32 :optional port endian
read-f64 :optional port endian

\{binary.io\} Reads 16, 32, or 64-bit floating point numbers, respectively. 32bit is IEEE754 single-precision, and 64bit is IEEE754 double-precision numbers. 16-bit floating point number consists of 1-bit sign, 5-bit exponent and 10-bit mantissa, as used in some HDR image format.

If port is omitted, current input port is used. If port reaches EOF before a complete number is read, EOF is returned.

write-f16 val :optional port endian
write-f32 val :optional port endian
write-f64 val :optional port endian

\{binary.io\} Writes a real number val to port in 16, 32, or 64-bit floating point number, respectively. If port is omitted, current output port is used.

I/O using uniform vectors

In the following routines, the argument uv can be any type of uniform vector; if it is not a u8vector, it is treated as if (uvector-alias <u8vector> uv) is called—that is, it reads directly from the memory image that holds the uvector’s content. The pos argument specifies the byte position from the beginning of the memory area (it is always byte position, regardless of the uniform vector’s element size).

get-u8 uv pos :optional endian
get-u16 uv pos :optional endian
get-u32 uv pos :optional endian
get-u64 uv pos :optional endian
get-s8 uv pos :optional endian
get-s16 uv pos :optional endian
get-s32 uv pos :optional endian
get-s64 uv pos :optional endian
get-f16 uv pos :optional endian
get-f32 uv pos :optional endian
get-f64 uv pos :optional endian

{binary.io} Reads a number of a specific format from a uniform vector uv, starting at a byte position pos. An error is signaled if the specified position makes reference outside of the uniform vector’s content. Returns the read number.

get-u16be uv pos
get-u16le uv pos
get-u32be uv pos
get-u32le uv pos
get-u64be uv pos
get-u64le uv pos
get-s16be uv pos
get-s16le uv pos
get-s32be uv pos
get-s32le uv pos
get-s64be uv pos
get-s64le uv pos
get-f16be uv pos
get-f16le uv pos
get-f32be uv pos
get-f32le uv pos
get-f64be uv pos
get-f64le uv pos

{binary.io} These are big-endian (be) or little-endian (le) specific versions of get-* procedures. In speed-sensitive code, you might want to use these to avoid the overhead of optional-argument handling.

get-uint size uv pos :optional endian
get-sint size uv pos :optional endian

{binary.io} Read size octets from uvector uv, starting from pos-th octet, as an unsigned or signed integer, respectively.

(get-uint 3 '#u8(1 2 3 4) 1 'big-endian)
⇒ 131884 ; #x020304

(get-sint 3 '#u9(1 2 3 #xff) 1 'little-endian)
⇒ -64766 ; sign extended #xff0302

put-u8! uv pos val :optional endian
put-u16! uv pos val :optional endian
put-u32! uv pos val :optional endian
put-u64! uv pos val :optional endian
put-s8! uv pos val :optional endian
put-s16! uv pos val :optional endian
put-s32! uv pos val :optional endian
put-s64! uv pos val :optional endian
put-f16! uv pos val :optional endian
put-f32! uv pos val :optional endian
put-f64! uv pos val :optional endian

{binary.io} Writes a number val into a uniform vector uv in a specific format, starting at a byte position pos. An error is signaled if the specified position makes reference outside of the uniform vector’s content.
Chapter 12: Library modules - Utilities

put-u16be! uv pos val  [Function]
put-u16le! uv pos val  [Function]
put-u32be! uv pos val  [Function]
put-u32le! uv pos val  [Function]
put-u64be! uv pos val  [Function]
put-u64le! uv pos val  [Function]
put-s16be! uv pos val  [Function]
put-s16le! uv pos val  [Function]
put-s32be! uv pos val  [Function]
put-s32le! uv pos val  [Function]
put-s64be! uv pos val  [Function]
put-s64le! uv pos val  [Function]
put-f16be! uv pos val  [Function]
put-f16le! uv pos val  [Function]
put-f32be! uv pos val  [Function]
put-f32le! uv pos val  [Function]
put-f64be! uv pos val  [Function]
put-f64le! uv pos val  [Function]

{binary.io} These are big-endian (be) or little-endian (le) specific versions of put-* procedures. In speed-sensitive code, you might want to use these to avoid the overhead of optional-argument handling.

put-uint! size uv pos val :optional endian  [Function]
put-sint! size uv pos val :optional endian  [Function]

{binary.io} Write an unsigned or signed integer val into an uvector uv starting from pos-th octet, for size octets, respectively.

Compatibility notes

read-u8 etc. were called read-binary-uint8 etc., and read-f32 and read-f64 were called read-binary-float and read-binary-double, respectively. These old names are still supported for the backward compatibility but their use is deprecated. The reason of the changes is for brevity and for consistency with the uniform vectors.

12.2 binary.pack - Packing binary data

binary.pack  [Module]

This module provides an interface for packing and unpacking (writing and reading) binary data with templates. The functionality was inspired largely by the Perl pack/unpack functions, with comparison of similar features from other languages, however an effort was made to make it more general and more efficient, to be usable for database-like processing. To that end, the most notable differences are that any packable value is unpackable (and vice versa), and the default behavior is to pack and unpack using port I/O, so you can seek in a large file and unpack from it. Also, templates may be stored as dispatch closures to pack, unpack or even skip over values without re-parsing the template.

pack template list :key output to-string?  [Function]

{binary.pack} Writes the values in list to the current output port, according to the format specified by the string template. The template string is a series of single character codes, optionally followed by a numeric count (which defaults to 1). The format characters can generally be divided into string types, which interpret the count as a string byte size, and object types, which treat the count as a repetition indicator. The count may be specified as the character *, which means to use the full size of the string for string types, and use
all remaining values for object types. Counts may also be specified as a template enclosed in brackets, which means the count is the byte size of the enclosed template. For example, x[L] skips a long. The special format character / may be used to indicate a structure where the packed data contains a dynamic count followed by the value itself. The template is written as \langle count-item\rangle/\langle value-item\rangle, where \langle count-item\rangle is any template character to be interpreted as a numeric count, and \langle value-item\rangle is any other template character to use this count. If a normal count is given after \langle value-item\rangle it is ignored. The format character @ may be used with a count to pad to an absolute position since the start of the template. Sub-templates may be grouped inside parentheses. If angle-brackets are used, then they also behave as group operators but recursively operate on nested lists. The string types:

- **a** An arbitrary incomplete string, null padded.
- **A** A text string, space padded.
- **Z** A null terminated (ASCIZ) string, null padded.
- **b** A bit string (ascending bit order inside each byte).
- **B** A bit string (descending bit order inside each byte).
- **h** A hex string (low nybble first).
- **H** A hex string (high nybble first).

The object types:

- **c** A signed 8bit integer.
- **C** An unsigned 8bit integer.
- **s** A signed short (16 bit) value.
- **S** An unsigned short (16 bit) value.
- **i** A signed integer (>= 32 bit) value.
- **I** An unsigned integer (>= 32 bit) value.
- **l** A signed long (32 bit) value.
- **L** An unsigned long (32 bit) value.

- **n, n!** An unsigned and signed short (16 bit) in "network" (big-endian) order.
- **N, N!** An unsigned and signed long (32 bit) in "network" (big-endian) order.
- **v, v!** An unsigned and signed short (16 bit) in "VAX" (little-endian) order.
- **V, V!** An unsigned and signed long (32 bit) in "VAX" (little-endian) order.
- **q** A signed quad (64 bit) value.
- **Q** An unsigned quad (64 bit) value.
- **f** A single-precision float in the native format.
- **d** A double-precision float in the native format.

- **w** A BER compressed integer. An unsigned integer in base 128, most significant digit first, where the high bit is set on all but the final (least significant) byte. Thus any size integer can be encoded, but the encoding is efficient and small integers don’t take up any more space than they would in normal char/short/int encodings.

- **x** A null byte.
An sexp, handled with read and write.

If the optional keyword :output is given that port is used instead of the current output port. If :to-string? is given and true, then pack accumulates and returns the output as a string.

Note that the returned string may be an incomplete string if the packed string contains a byte sequence invalid as a character sequence.

```scheme
(pack "CCCC" '(65 66 67 68) :to-string? #t)
⇒ "ABCD"
```

```scheme
(pack "C/a*" '("hello") :to-string? #t)
⇒ "\x05hello"
```

**unpack template :key :input :from-string**  
{binary.pack} The complement of pack, unpack reads values from the current input port assuming they’ve been packed according to the string template and returns the values as a list. unpack accepts the same format strings as pack. Further, the following tautology holds:

```scheme
(equal? x (unpack fmt :from-string (pack fmt x :to-string? #t)))
```

for any list x and format string fmt. The only exceptions to this are when the template includes a * and when the o template is used, since Scheme numeric literals cannot be reliably delimited (though future versions of pack may circumvent this by registering a new read syntax).

If the optional keyword :input is given that port is used instead of the current input port. If :from-string is given, then pack reads input from that string.

```scheme
(unpack "CCCC" :from-string "ABCD")
⇒ '(65 66 67 68)
```

```scheme
(unpack "C/a*" :from-string "\x05hello")
⇒ '("hello")
```

Note: in the current version, @ in unpack template has a bug and does not work as supposed. It will be fixed in the future version.

**unpack-skip template :key :input**  
{binary.pack} unpack-skip is the same as unpack except it does not return the values. In some cases, particularly with fixed-size templates, this can be much more efficient when you just want to skip over a value.

**make-packer template**  
{binary.pack} The low-level interface. This function returns a dispatch closure that can be used to pack, unpack and skip over the same cached template. The dispatch closure accepts symbol methods as follows:

```scheme
'pack list
pack the items in list to the current output port.

'unpack
unpack items from the current input port.

'skip
skip items from the current input port.

'packer
return the cached 'pack closure

'unpacker
return the cached 'unpack closure.

'skipper
return the cached 'skip closure.

'length
return the known fixed length of the template.
'variable-length?
    return #t if the template has variable length elements.

12.3 compat.chibi-test - Running Chibi-scheme test suite

compat.chibi-test
    [Module]
    Quite a few srfis come with test suites that’s to be run with Chibi Scheme test framework. This module enables Gauche to run the test code as is.

chibi-test code ...
    [Macro]
    {compat.chibi-test} Run code . . ., while translating Chibi test framework to Gauche’s.
    A typical usage is to write a wrapper that includes the original test code (suppose it’s called test-suite.scm):

        (use gauche.test)
        (test-start "running test-suite.scm")
        (chibi-test
          (include "test-suite.scm")))
        (test-end)

Chibi’s test directives are translated to Gauche’s test directives (see Section 9.34 [Unit testing], page 521, for Gauche’s test framework).

The main thing is that Chibi allows expressions and definitions to be intermingled within a body, while Gauche only allows all definitions before expressions within a body. We expand such body into nested let by chibi-test macro. Chibi test macros (e.g. test-assert) are defined as local macros in chibi-test expansion, which expand into gauche.test macros.

Note that we ignore use forms inside chibi-test; we might want to use different modules that work better in Gauche. Necessary modules need to be use’d before you call chibi-test.

You may want to check out test/srfi.scm in Gauche source tree for the use case.

12.4 compat.norational - Rational-less arithmetic

compat.norational
    [Module]
    Until release 0.8.7, Gauche didn’t have exact rational numbers. It was able to read the rational number literals such as 2/3, but they are immediately coerced to inexact real numbers (except when it represents a whole integer). And if you divided an exact integer by another exact integer, the result could be coerced to an inexact real if the result wasn’t a whole integer.

As of 0.8.8, this is not the case anymore. Exact division always yields exact result, except when the divisor is zero.

    (/ 2 3)  ⇒ 2/3
    (/ 5)    ⇒ 1/5
    (/ 4 2)  ⇒ 2

This is more precise, but has one drawback: exact rational arithmetic is much slower than the integer and inexact real arithmetic. If you inadvertently produce a rational number in the early stage of calculation, and continue to apply exact arithmetic, performance would be degraded miserably.

The proper way to solve this is to insert exact->inexact to appropriate places. However, to ease the transition, you can just import this module and the division / behaves in the way it used to.

    (use compat.norational)

    (/ 2 3)  ⇒ 0.6666666666666666
The effect is not global, but only to the modules you explicitly import `compat.norational`. This module only redefines `/`. So if your code has exact rational literals, they are treated as exact rationals rather than coerced to inexact reals. You should prefix rational literals with `#i` to force Gauche to coerce them to inexact reals:

```
gosh> 1/3
1/3
```
```
gosh> #i1/3
0.3333333333333333
```

### 12.5 `compat.real-elementary-functions` - Backward-compatible real elementary functions

*compat.real-elementary-functions* [Module]

Gauche 0.9.12 and earlier had undocumented real-only elementary functions, such as `%sin` and `%exp`. They were intended to be a building block of the complex elementation functions, hence prefixed with `%`. However, they were visible in `gauche` module, and sometimes used directly because they were fast.

SRFI-94 introduced names for real-only elementation functions such as `real-sin`, so we changed built-in `%sin` etc. to follow that names. See Section 6.3.4 [Arithmetics], page 126, for those real-only elementary functions.

This module is to ease migration from the legacy code that uses old names. Just add the following form at the beginning of old code and the source would work with both versions.

```scheme
(cond-expand
  [(library (compat real-elementary-functions))
   (use compat.real-elementary-functions)]
  [else])
```

New code must use `real-sin` etc. instead of the names provided here.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%exp</code> <code>x</code></td>
<td>[Function] Replacing preceding <code>%</code> with <code>real-</code></td>
</tr>
<tr>
<td><code>%sin</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%cos</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%tan</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%sinpi</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%cospi</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%tanpi</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%asin</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%atan</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%sinh</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%cosh</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%tanh</code> <code>x</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>%expt</code> <code>x</code> <code>a</code></td>
<td>[Function]</td>
</tr>
</tbody>
</table>

{compat.real-elementary-functions} These are the same as the built-in procedures replacing preceding `%` with `real-`; e.g. `%exp` ≡ `real-exp`. See Section 6.3.4 [Arithmetics], page 126, for the description of built-in procedures.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%log</code> <code>x</code> <code>optional base</code></td>
<td>[Function]</td>
</tr>
</tbody>
</table>

{compat.real-elementary-functions} This is a real-only logarithm of `x` base `base`. `x` must be a non-negative real number, and `base` must be a positive real number. If `base` is omitted,
Napier’s number is assumed, that is, it returns a natural logarithm of \( x \). If \( x = 0 \), \(-\infty\) is returned.

Gauche does not have built-in `real-log`, to avoid conflict with SRFI-94’s `real-log` which takes `base` argument first. For real-only natural logarithm, Gauche has `real-ln`.

See Section 10.3.24 [R7RS flonum], page 670, which defines flonum-only version of logarithms such as `fllog`, `fllog2`, and `fllog10`.

### 12.6 control.cseq - Concurrent sequences

`control.cseq` Concurrent sequence (cseq) is a lazy sequence (see Section 6.18.2 [Lazy sequences], page 241), but the generator runs in a separate thread. You can use producer-consumer pararellism very easily using cseq; from the consumer side, it just looks like an ordinary list. Synchornization is implicitly taken care of.

Internally, it uses mtqueue (see Section 12.19 [Queue], page 837) for synchronization.

```
control.cseq gen :key queue-length
{control.cseq} Create a lazy sequence from generator, much like generator->lseq, except that gen runs in a separate thread.
```

The returner value looks like an ordinary list, but its cdrs are computed in parallel. If a cdr isn’t computed yet, the reader thread waits until a value becomes available. The generator `gen` can be called to generate values asynchronously, until the internal queue gets full. (Compare this to an ordinary lseq, in which `gen` is called when it is required, and it runs in the same thread as the caller.)

If `gen` raises an error, it is caught, and reraised when the consumer reads to the point when `gen` raised the error.

The optional `queue-length` must be a nonnegative exact integer or `#f`. If it is an integer, it specifies the length of the internal queue. If it is `#f`, an appropriate value is selected by the library (currently 64).

Note that Gauche’s lseq read-ahead one item (see Section 6.18.2 [Lazy sequences], page 241). So even if you set the queue length to 0, `gen` is called before the consumer reads out any value.

```
coroutine->cseq proc :key queue-length
{control.cseq} Returns an lseq, whose value is generated in proc, which is called in a separate thread. (See also coroutine->lseq, Section 9.15.1 [Lazy sequence constructors], page 451). The proc argument is a procedure to be called with one argument, yield. The yield argument is a procedure, and whenever it is called with a value, that value becomes the next item of the resulting lseq. When proc returns, it becomes the end of the lseq.
```

If `proc` raises an error, it is caught, and reraised when the consumer reads to the point when previously generated values are all read and next value is about to read.

The optional `queue-length` must be a nonnegative exact integer or `#f`. If it is an integer, it specifies the length of the internal queue. If it is `#f`, an appropriate value is selected by the library (currently 64).

### 12.7 control.future - Futures

`control.future` A future is a simple construct for concurrent computation.

It encloses an expression, and compute its value concurrently. The result of computation can be retrieved later with `future-get`.
Futures are introduced in MultiLisp, in which retrieval of the computed value is implicit—a future is substituted with the result automatically. Racket and Guile have futures as a library, though the primitive to retrieve the result is called touch. We avoided to use the name since touch is too generic.

```scheme
(future expr)  {control.future} Returns a future object, which run the computation of expr in a separate thread. The result(s) of expr can be retrieved by future-get. Note that expr can yield multiple values.

The expr is evaluated in the same environment as future appears, though if an exception raised within expr is not caught, its delivery is delayed until you call future-get.

The following example runs HTTP access concurrently with other computation, and retrieves the result later.

```scheme
(use control.future)
(use rfc.http)

(let1 f (future (http-get "example.com" "/"))
  ... some computation ...
  (receive (code headers body) (future-get f)
    ...))
```

```scheme
(make-future thunk)  {control.future} Returns a future that calls thunk in a separate thread.

```scheme
(future expr) ≡ (make-future (lambda () expr))
```

```scheme
(future? obj)  {control.future} Returns #t if obj is a future, #f otherwise.
```

```scheme
(future-get future :optional timeout timeout-val)  {control.future} The argument must be a future. Retrieve the result(s) of future.

If the result of future is already available, it is returned immediately. If future is still computing, future-get blocks until the result is ready by default. You can limit how long you will wait by timeout argument, which can be #f (default, no timeout), a nonnegative real number (relative time in seconds), or a <time> object (absolute timepoint). If the timeout reaches before the result is available, timeout-val is returned, which defaults to #f. Calling future-get more than once returns the same result.

If an uncaught exception is raised during computation in the future, it is kept and reraised from future-get. It is handled in the dynamic environment of future-get (not the one in the original future call). If you call future-get again on such future, the effect is undefined (currently it returns #<undef> without raising an exception, but it may change in future).

```scheme
(future-done? future)  {control.future} Returns #t if computation in future is finished, #f otherwise.
```

12.8 control.job - A common job descriptor for control modules

```scheme
control.job  {Module}
```

This module provides a job record type, a lightweight structure to be used in the control flow subsystems (control.* modules). Currently the only user is control.thread-pool, but some other modules are planned to use job records.
A job record may be returned to an application by other control.* modules so that the application can keep track of the job. It’s not meant for general use, however. An application isn’t supposed to create a new job, or to modify its content; it can just query the job’s properties.

In this section we only describe procedures an application needs to know. The interface for control subsystems is still fluid and may be changed as more subsystems are developed.

Different control flow subsystems may use job structure differently. This section only describes the common properties. Check the individual control flow module to know how to handle returned job objects.

```plaintext
job [Record type]
{control.job} A record type denotes the job. Applications should treat it as an opaque structure.

job? obj [Function]
{control.job} Returns #t iff obj is a job record, #f otherwise.

job-status job [Function]
{control.job} Returns the status of the job. It may be either one of the followings.

#f Newborn or orphaned job. Usually an application won’t see a job in this status.
acknowledged A job is recognized by a control flow library, but haven’t yet been run.
running A job is being processed.
done A job is finished. An application can retrieve its result by job-result.
error A job is terminated by an error. An application can retrieve the error causing condition by job-result.
killed A job is killed by external force. An application can retrieve the reason of kill (which is specific to a particular control flow subsystem) by job-result.

job-result job [Function]
{control.job} If the job is in done status, it returns the result of the job. If the job is in error status, it returns the condition object that describes the error. If the job is in killed status, it returns an object describing the reason of kill. The details of the object depends on a particular control flow library. Calling job-result on a job in any other status may return anything; you can’t rely on the result.

job-wait job :optional timeout timeout-val [Function]
{control.job} Suspends the calling thread until the job becomes either done, error or killed status. If the job is already in one of those status, it returns immediately. Returns job’s status.

If timeout is given and not #f, it must be a valid timeout spec (a <time> object that represents an absolute time point, or a real number that represents a relative time in seconds.) The meaning of timeout is the same as in mutex-unlock! (see Section 9.35.4 [Synchronization primitives], page 536). Once the timeout reaches, job-wait returns no matter how the job’s status is, and returns the value specified to timeout-val, which defaults to #f.

Depending on the control flow subsystem, jobs created by it may not be waitable; check out each subsystem’s documentation for the details.
```
job-acknowledge-time job
job-start-time job
job-finish-time job

{control.job} If the control flow subsystem keeps track of timestamps, these procedures returns the time (in <time> objects) when the job is acknowledged, started and finished (either normally, or abnormally by an error or by being killed). If the job hasn’t reached to certain status, #f is returned instead.

If the subsystem does not track timestamps, these procedures always returns #f.

12.9 control.plumbing - Plumbing ports

control.plumbing

This module provides utilities to use ports as communication channels between threads. Although some use cases don’t involve threads, the general mechanism implicitly use threads. For example, a pump connects an input port and an output port, making the data available from the input port to be written to the output port. It spawns a thread to read from the given input.

With this module, you can construct a system that takes data from ports, and delivers it to other ports. It is like you construct plumbing system with multiple inlets and outlets; hence we named it plumbing.

See make-plumbing below for the complete specification of plumbing.

Plumbing system

make-plumbing

{control.plumbing} Creates and returns a new plumbing.

To function, a plumbing need to have one or more inlets and one or more outlets. Data flowing in from inlets will flow out to outlets.

Each inlet/outlet port can be either an input port or an output port; so we’ll have the following four cases:

Inlet input port
An input port that provides data from a producer. A dedicated thread is spawned to read and handle the data from this port.

Inlet output port
An output port, to which a producer can write data. Data written to this port will be available to outlet ports.

Outlet input port
An input port from which a consumer can read data. It has an internal buffer to keep data flowed in from inlets.

Outlet output port
An output port to which the data will be written.

All read/write operations are mutexed, so you can use a plumbing as a communication channel between threads.

The following example connects subprocess’s stdout and stderr into the current output port. Both output of the subprocess is automatically read from the pipe and sent to the current output port as produced.

(use gauche.process)
(use control.plumbing)
(let ([p (run-process '(ping "127.0.0.1") :output :pipe :error :pipe)]
    [plumbing (make-plumbing)])
  (add-inlet-input-port! plumbing (process-output p))
  (add-inlet-input-port! plumbing (process-error p))
  (add-outlet-output-port! plumbing (current-output-port)))

For typical plumbing configurations, there are utility procedures defined so that you don’t need to connect pots manually; see “Preconfigured plumbings” section below.

**plumbing? obj**

{control.plumbing} Returns #t if obj is a plumbing, #f if not.

**plumbing-inlet-ports plumbing**

{control.plumbing} Returns a list of ports connected to the plumbing’s inlets and outlets, respectively.

**plumbing-outlet-ports plumbing**

{control.plumbing} Returns a list of ports connected to the plumbing’s inlets and outlets, respectively.

**plumbing-get-port plumbing name**

{control.plumbing} The name argument must be a symbol. Returns a port connected to plumbing and with name. Returns #f if there’s no such ports.

**port-plumbing port**

{control.plumbing} A port opened for a plumbing with open-outlet-input-port, open-inlet-output-port, or other utility procedures, keeps which plumbing it belongs to. This procedure returns the plumbing associated with port. If port is not made from a plumbing, #f is returned.

Note that an existing port added to a plumbing by add-inlet-output-port! etc. does not keep the plumbing it is attached to, and this procedure returns #f on such a port.

### Creating and attaching ports to plumbing

**open-inlet-output-port plumbing :optional name**

{control.plumbing} Creates an output port connected to plumbing’s inlet; that is, the data written to the port will flow to plumbing’s outlets.

The optional name should be a symbol or #f. If it is a symbol, the port will be retrieved from plumbing with plumbing-get-port.

Applying port-plumbing on the returned port returns plumbing.

Once you close the returned port, the port is removed from plumbing. Once all inlet ports are closed, the outlet ports of plumbing get EOF.

**add-inlet-input-port! plumbing iport :optional name**

{control.plumbing} Connects an existing input port iport to plumbing’s inlet. A thread in plumbing reads from iport and feeds the data to its outlets.

The optional name should be a symbol or #f. If it is a symbol, the port will be retrieved from plumbing with plumbing-get-port.

Once EOF is read from iport, it is removed from plumbing. Once all inlet ports are closed, the outlet ports of plumbing get EOF.

**open-outlet-input-port plumbing :optional name**

{control.plumbing} Creates an input port connected to plumbing’s outlet; that is, from that port you can read the data fed to plumbing’s inlets.

The optional name should be a symbol or #f. If it is a symbol, the port will be retrieved from plumbing with plumbing-get-port.
Applying `port-plumbing` on the returned port returns `plumbing`.

Once all inlets are closed, the returned port yields EOF. You can close the returned port at any time; then the port is removed from `plumbing`’s outlets and no data will be delivered to it.

**add-outlet-output-port!**

```lisp
(plumbing oport :optional name :key]
  close-on-eof asynchronous
{
  control.plumbing} Connects an existing output port `oport` to `plumbing`’s outlet. Data flowed in from `plumbing`’s inlet will be written to `oport`.

The optional `name` should be a symbol or `#f`. If it is a symbol, the port will be retrieved from `plumbing` with `plumbing-get-port`.

You can close the returned port at any time; then the port is removed from `plumbing`’s outlets and no data will be delivered to it.

By default, `oport` is left as is after all inlets are closed; `plumbing` won’t write anything to `oport` but you can still write to it. If you specify a true value to the `close-on-eof` keyword argument, however, `plumbing` will close `oport` when all of its inlets are closed. It is useful, for example, when `oport` is a file output port and you want to close it once `plumbing` is done. However, if `oport` is a standard output port for logging purpose, you don’t want it to be automatically closed.

If a true value is given to the `asynchronous` keyword argument, a thread is spawn to feed the data to `oport`; it is useful if `oport` can be blocked, e.g. an OS pipe or a socket. Without `asynchronous`, blocked output to `oport` causes data flow to inlets to be blocked as well. With `asynchronous`, other threads can keep feeding to inlets.

**plumbing**

```lisp
(plumbing spec . . .)
{
  control.plumbing} A utility plumbing constructor. It creates a new plumbing, and adds inlets and outlets as specified by `specs`, then returns the plumbing. Each `spec` must be one of the following forms. In the description, `iport` is an input port, `oport` is an output port, and `name` is a symbol.

`(< iport)`
`(< iport name)`
  Adds `iport` to the plumbing’s inlet, optionally naming it with `name`.

`(< name)`
  Creates an input output port with name `name`. The output port can be retrieved by `plumbing-get-port`.

` (> oport)`
` (> oport name)`
` (> oport (option . . .))`
` (> oport name (option . . .))`
  Adds `oport` to the plumbing’s outlet, optional naming it with `name`. Options can be one of the keywords :close-on-eof, :coe, asynchronous, and :async. :close-on-eof and asynchronous has the effect of giving true to those keyword arguments to `add-outlet-output-port!`. :coe and :async are synonyms of :close-on-eof and :asynchronous, respectively.

` (> name)`
  Creates an outlet input port with name `name`. The output port can be retrieved by `plumbing-get-port`.

**Preconfigured plumbings**

The following are convenience procedures to construct typical plumbings.
**make-pipe**: `:key` `num-inlets` `num-outlets`  
{control.plumbing} A pipe is a passive plumbing device that has inlet output port(s) and outlet input port(s). Data written to the inlet output ports will be buffered and available to read from the outlet input ports. This procedure creates a pipe with `num-inlets` inlet output ports and `num-outlets` outlet input ports, and returns two values, a list of inlet ports and a list of outlet ports.

Both `num-inlets` and `num-outlets` should be a positive exact integer. If omitted, 1 is assumed.

If there are more than one outlets, data is duplicated; each consumer reads the same data. Buffers are attached to each outlet, so each consumer can read the outlet in its own pace.

The created plumbing can be obtained by applying `port-plumbing` to any of the returned ports.

This procedure is similar to `sys-pipe`, which creates a pair of ports using OS's pipe mechanism (see Section 6.24.4.5 [Other file operations], page 304). However, the pipe device created by `make-pipe` is a user-land construct and only for inter-thread communication; it can’t be used for inter-process communication. On the other hand, you can have n-in m-out pipe using `make-pipe`.

**make-pump**: `inlet-iports` `outlet-oports`  
{control.plumbing} A pump is an active plumbing device that reads data from the given inlet input ports, and write it out to the given outlet output ports.

The `inlet-iports` argument must be a list of input ports, and `outlet-oports` argument must be a list of output ports.

A dedicated thread is spawn to each inlet input port to read from it.

**open-broadcast-output-port**: `oport` . . .  
{control.plumbing} All the arguments must be output ports. Returns an output port such that data written to it go to every `oport`. Internally, it creates a plumbing, adds `oports` to its outlet, and returns a new inlet output port. The plumbing can be retrieved by calling `port-plumbing` on the returned port.

This is Common Lisp’s `make-broadcast-stream`.

**open-tapping-input-port**: `inlet-iport` `outlet-oport` :`key` `close-on-eof`  
{control.plumbing} It creates a pump, a plumbing that reads from `inlet-iport` and writes out the read data to `outlet-oport`. Additionally, it creates and returns an outlet input port, from which you can read the data flowing in the pump.

An example use case is to monitor process output. You spawn a process with its stdout connected to a pipe, then call (open-tapping-input-port (process-output subprocess) (current-output-port)). With this, the subprocess’s output automatically flows to your process’s output. However, that data is also available from the returned input port, and you can check it.

Note that the data passed from `inlet-iport` to `outlet-oport` are buffered for the returned input port until you read from it. If you’re done with monitoring, you can close the returned input port safely; the data will no longer be buffered, but the pump will keep running until the input reaches EOF.

The pump plumbing is available by calling `port-plumbing` on the returned input port.

### 12.10 control.pmap - Parallel map

**control.pmap**  
This module provides high-level utilities to run code in parallel using threads. For example, the `pmap` procedure applies the given procedure on each elements in the collection and gathers the results into a list, just like `map`, but the application of the procedure is done in parallel.
A desired parallelization strategy differs for application, so we also provide *mapper* objects, that encapsulate how the work is distributed.

**High-level API**

`pmap proc collection :key mapper`  [Function]

{control.pmap} The *proc* argument must be a procedure that takes one argument, and *collection* must be a collection (see Section 9.6 [Collection framework], page 404).

Apply *proc* on each element of *collection*, possibly concurrently using multiple threads. The result is gathered into a list and returned.

You can pass a mapper to the *mapper* keyword argument to specify how the task is distributed to multiple threads.

`pfind pred collection :key mapper`  [Function]

`pany pred collection :key mapper`  [Function]

{control.pmap} These are to be used to find one element that satisfies the predicate *pred*. As soon as the element is found, other tasks are cancelled.

`pfind` is like *find* that returns the element that satisfies *pred*, while *pany* is like *any* that returns the result of *pred* that isn’t #f.

If no element satisfies *pred*, #f is returned.

If there are more than one element that satisfy *pred*, which one is picked depends on various factors, so you shouldn’t count on a deterministic behavior.

**Mappers**

A mapper is an object that encapsulates a strategy to run tasks in parallel. We provide the following mappers.

**Static mapper**

Creates several threads and distribute the tasks evenly. It is suitable when the number of tasks are large and each task is expected to take mostly same amount of time, for it takes less overhead than other multi-threading mappers.

**Pool mapper**

Uses thread pool to process the tasks. It is suitable when the number of tasks are large and/or the execution time of each task varies a lot. You can also reuse the pooled threads, so that you can reduce the overhead of thread creation.

**Fully concurrent mapper**

Creates one thread per each task. It is suitable when the task involves blocking I/O calls, and the number of tasks are not so large.

**Sequential mapper**

This runs tasks sequentially in a calling thread. No concurrency involved. It serves two purposes: (1) On a single-core system, this is the least overhead strategy, and (2) You can test the algorithmic correctness without complication of concurrency. On single-core systems, this mapper is the default value of `default-mapper`.

`default-mapper`  [Parameter]

{control.pmap} A parameter keeping a mapper to be used by `pmap` etc. when no mapper is specified.

The default is a static mapper (with the number of threads same as the number of available cores) if Gauche is running system with more than one core, or a sequential mapper otherwise. The mapper set to this parameter is reused, or even is used simultaneously from multiple `pmap` calls. Pool mappers with external pool keeps a given thread pool in it, so you should be careful to use such mapper as the default mapper.
sequential-mapper
{control.pmap} Returns a singleton instance of the sequential mapper.

make-static-mapper :optional num-threads
{control.pmap} Returns a new instance of a static mapper, which spawns num-threads threads on execution, each of which handles evenly divided tasks. This mapper is suitable if you have large number of small tasks with even load.

If num-threads is omitted, the number of available processors returned by sys-available-processors is used (see Section 6.24.3 [Environment inquiry], page 295).

make-pool-mapper :optional external-pool
{control.pmap} Returns a new instance of a pool mapper, which uses a thread pool (see Section 12.12 [Thread pools], page 826) to run the tasks. It is suitable when the load of tasks varies a lot.

If external-pool is not given, the mapper creates a thread pool, and shut it down, every time high-level mapping operation is called. This usage is local; that is, the thread pool is contained within one call of pmap etc., and won’t be shared.

Alternatively, you can pass an existing thread pool to external-pool to be used. The pool will be reused every time you use this mapper instance. Using an external pool will eliminate overhead of thread pool creation and shutting down every time you run pmap; however, you have to be aware of those caveats:

- It’s your responsibility to shut down the thread pool after you’re done with the mapper.
- The mapper keeps the given thread pool and reuses it every time it is passed to pmap etc., so you have to make sure that one mapper is not used simultaneously in more than one pmap etc.. Be careful using this type of pool mapper as the default mapper.

make-fully-concurrent-mapper :optional timeout timeout-val
{control.pmap} Returns a new instance of a fully-concurrent mapper, which spawns as many threads as the elements in the given collection to perform the operation concurrently. It is suitable when you don’t have many tasks, but each task may perform blocking I/O calls. The overhead of creating threads are relatively large, but you may be able to utilize CPU more while most of the threads are waiting I/O.

The optional timeout and timeout-val arguments are passed to thread-join! (see Section 9.35.2 [Thread procedures], page 532). It is useful when I/O operations may take too long and you want to guarantee the entire operation finishes within certain time limit.

12.11 control.scheduler - Scheduler

control.scheduler
[Module]

A scheduler is a device to run tasks in scheduled time.

Each scheduler manages one or more tasks. Each task has attached time to run. A task can be run once, or can be run periodically.

<scheduler>
{control.scheduler} A device to run tasks in scheduled time. Each instance of this class has its own thread, and maintains a list of tasks along the information when each task should be run. Tasks can be registered by scheduler-schedule!.

If a task throws an error, a procedure bound to error-handler slot is invoked with the thrown condition as the only argument. If no error-handler is registered, or error-handler throws an error again, then the scheduler stops. The thrown condition is kept in the scheduler and can be retrieved with scheduler-terminate!.
error-handler

This slot can be initialized with :error-handler keyword argument. It must be either #f or a procedure that takes one argument. If it is a procedure, it is invoked when a task throws an error, and the argument is the raised condition. The result of error-handler procedure is discarded.

scheduler-running? scheduler
{control.scheduler} Returns #t iff scheduler is running.

scheduler-schedule! scheduler thunk when :optional interval
{control.scheduler} Inserts a new tasks that runs thunk into scheduler. The task can be run once, or periodically.

The when argument specifies when the task should first run. It can be either one of the followings:

<time> object of type-utc or time-tai type
    Specifies the absolute point of time.

<time> object of time-duration type
    Specifies the relative time since this procedure is called.

real number
    Specifies the relative time in seconds since this procedure is called.

The optional interval argument can be #f, real number or <time> object of time-duration type. If it is #f or 0, the task is one-shot, that is, not repeated. Otherwise, the task is repeated with the specified interval—the real number specifies the number of seconds.

Returns an integer that identifies the task. The task id can be used to cancel or reschedule the task.

scheduler-reschedule! scheduler task-id when :optional interval
{control.scheduler} Change the schedule of the task specified by task-id in the scheduler.

The meaning of when and interval argument is the same as scheduler-schedule!.

The same types of values as scheduler-schedule! are allowed for when and interval arguments, as well as a symbol unchanged, which indicates that value is left alone by the operation. Note that omitting interval means task will be non-repeating. If you want to change just the next time the repeating task is executed, pass unchanged to the interval.

If the scheduler doesn’t have a task with task-id, an exception is raised.

scheduler-remove! scheduler task-id
{control.scheduler} Remove the task specified by task-id from the scheduler. If the task is actually removed, #t is returned. If the scheduler doesn’t have a task with task-id, #f is returned.

scheduler-exists? scheduler task-id
{control.scheduler} Returns #t iff the scheduler has the task with task-id in the queue. Note that once the task is executed and not repeating, the task is removed from the queue.

scheduler-terminate! scheduler :key on-error
{control.scheduler} Stop the scheduler. Tasks still in the scheduler’s queue won’t executed, and no new task will be accepted. Once the scheduler is terminated, it can’t be restarted.

If no task has raised an exception, or all exceptions are handled by the error-handler of the scheduler, this procedure returns #t after the scheduler is stopped.
If any task has raised an exception and not handled by the error-handler, the behavior depends on the on-error keyword argument, which should be one of the following:

**:reraise**  The exception is reraised from scheduler-terminate! . This is the default.

**:return**  The exception is returned from scheduler-terminate! .

### 12.12 control.thread-pool - Thread pools

**control.thread-pool**  
Provides thread pools. Only available when Gauche is compiled with pthreads support.

**<thread-pool>**  
{control.thread-pool}  
A class for thread pool objects. It maintains a set of worker threads, and let them work on the jobs you ask to do asynchronously.

Currently the size of pool (number of threads) is fixed and you have to specify it when creating a pool. In future we might add a feature to grow or shrink the pool.

You can also set maximum backlog of the job queue. You cannot put a job when the queue already reaches the max length (see **add-job!** below).

**<thread-pool-shut-down>**  
{control.thread-pool}  
A condition indicating that a thread pool is already shut down by terminate-all! and no longer accepting new jobs. Inherits <error>. The following slot is provided.

  pool  
  [Instance Variable of <thread-pool-shut-down>]  
  The thread pool object that caused the condition.

**make-thread-pool size :key (max-backlog 0)**  
{control.thread-pool}  
Creates a new thread pool of size size (the number of worker threads). Optionally you can give a nonnegative integer to the maximum backlog; 0 means unlimited.

**thread-pool-results pool**  
{control.thread-pool}  
When you put a job to a thread pool, you can specify whether you need to check its result or not. If you say you need a result, the terminated job is queued to a result queue, an <mt-queue> object, in the pool. This procedure returns the pool's result queue. See Section 12.19 [Queue], page 837, for the details of <mt-queue>.

**thread-pool-shut-down? pool**  
{control.thread-pool}  
Returns #t if the thread pool is shut down and no longer accepting new jobs, or #f otherwise.

**add-job! pool thunk :optional (need-result #f) (timeout #f)**  
{control.thread-pool}  
Add a thunk to be executed in the thread pool pool. Returns a job record (see Section 12.8 [A common job descriptor for control modules], page 817).

The returned job record is not waitable; if you need to track its result, you have to give a true value to need-result argument. Then when the job is terminated (either normally or abnormally) the job is queued to the result-queue of the pool, and you can check the queue.

If you don’t pass a true value to need-result, the job won’t be queued to result-queue even it is terminated.

The returned job is timestamped. You can examine acknowledged time, start time and finish time of the job (if the job hasn’t been started and/or finished, the corresponding timestamp fields are #f.) It’s sometimes handy to find out how long the job was waiting in the queue and how long it took to run.
If the pool has positive max-backlog value, and it already has that many jobs to be waiting, then \texttt{add-job!} blocks until some jobs are start being executed. You can give a real number in seconds, or a \texttt{<time>} object as an absolute point of time, to the \texttt{timeout} argument to set the time limit of blocking. If timeout is reached, \texttt{add-job!} returns \texttt{#f} without creating any job. Omitting \texttt{timeout} or giving \texttt{#f} to it sets no timeout.

(Note: This behavior is different from 0.9.1, in which \texttt{add-job!} didn’t take the timeout argument and always behaved as if zero timeout value was given. To achieve the same behavior, you have to give 0 to the \texttt{timeout} argument explicitly.)

If the thread pool is shut down, this procedure raises \texttt{<thread-pool-shut-down>} condition.

\textbf{wait-all} \hspace{1em} \texttt{pool :optional \hspace{1em} \texttt{(timeout \hspace{1em} #f \hspace{1em} \texttt{(check-interval \hspace{1em} #e5e8}))}} \hspace{1em} \textbf{[Function]} \hspace{1em} \texttt{\{control.thread-pool\}} \hspace{1em} Wait for the job queue to be empty and all worker threads to finish. It is done by polling the pool’s status in every \texttt{check-interval} nanoseconds. Returns \texttt{#t} if all jobs are finished.

You can give a real number in seconds, or a \texttt{<time>} object as an absolute point of time, in \texttt{timeout} optional argument. When timeout is reached, \texttt{wait-all} returns \texttt{#f}.

While this procedure is called, no new jobs should be put into \texttt{pool}.

\textbf{terminate-all!} \hspace{1em} \texttt{pool :key \hspace{1em} \texttt{(force-timeout \hspace{1em} #f \hspace{1em} (cancel-queued-jobs \hspace{1em} #f))}} \hspace{1em} \textbf{[Function]} \hspace{1em} \texttt{\{control.thread-pool\}} \hspace{1em} Wait for all the queued jobs to be finished, then ask all threads to terminate. After calling this procedure, the pool no longer accepts new jobs. Calling \texttt{add-job!} on this module would raise a \texttt{<thread-pool-shut-down>} condition. This is intended to be called when shutting down the application.

By default, this procedure first waits for all queued jobs to be handled, then tries to terminate threads gracefully.

Giving a true value to the \texttt{cancel-queued-jobs} argument immediately cancels queued but not started jobs; the status of such jobs is set killed. It does not cancels already started jobs, though.

If you want to cancel already started jobs, you can give a timeout value (either \texttt{<time>} object to specify absolute point of time, or a real number indicating relative time in seconds) to the \texttt{force-timeout} argument. Once timeout is reached, it forcefully terminates the threads and the jobs handled at that time are also killed.

Forcing termination of threads is an extreme measure; the terminated thread may not have a chance to clean up properly. So it is usually better to give some time for the thread to finish the executing jobs.

\subsection*{12.13 crypt.bcrypt - Password hashing}

\textbf{crypt.bcrypt} \hspace{1em} \textbf{[Module]}

This module implements a password hashing algorithm using blowfish, and compatible to OpenBSD’s bcrypt algorithm (version 2a, 2b).

Don’t use version “2a” for new code. It’s vulnerable. Use version “2b”.

The typical usage of this module is simple enough. To get a new password hash value (e.g. for a new user), pass the password string to \texttt{bcrypt-hashpw} as the only argument:

\begin{verbatim}
(bcrypt-hashpw password)  ⇒ hashed-string
\end{verbatim}

The routine automatically adds a salt value. The returned hash string can be stored in the user database. To check if the given password matches the stored one, pass the hashed string as the second argument of \texttt{bcrypt-hashpw} to check the password.

\begin{verbatim}
(bcrypt-hashpw password hashed-string)
\end{verbatim}
If the given password is correct, the returned value should exactly matches hash-string.

**bcrypt-hashpw** *password :optional setting* [Function]
{crypt.bcrypt} Calculates a hash value of *password*, using the salt value and parameters included in *setting*. If *setting* is omitted, a suitable default settings and random salt value is chosen automatically.

The returned hash value contains the salt value and parameters, and can be used as *setting*. So, to check the password against existing hash value, just pass the hash value to *setting*; if the password is correct, the returned hash value should match the one you passed in.

The bcrypt algorithm supports up to 72 octets for the password.

To tweak parameters when you calculate a new hash value, use **bcrypt-gensalt** below to get the initial *setting* value.

**bcrypt-gensalt** *key prefix count entropy-source* [Function]
{crypt.bcrypt} Returns a string that contains given parameters and suitable to pass to the *setting* argument of **bcrypt-hashpw**.

The *prefix* argument specifies the version/scheme of password hashing. Currently $2a$ and $2b$ are supported, which means the blowfish algorithm compatible to bcrypt. But $2a$ is vulnerable. Use $2b$ for new code. If you omit *prefix*, use $2b$ for default value.

The *count* argument specifies the amount of iterations; the larger the value is, the more time is required to calculate the hash value. Note that for the password hashing, taking more time is actually a good thing, for it works against the dictionary attack. For normal password checking you need to run the hash routine only once per login, so it doesn’t matter if the calculation takes a fraction of second. The bcrypt algorithm iterates (expt 2 *count*) times.

The *entropy-source* argument is a u8vector to feed a random bytes. For bcrypt algorithm it must be at least 16 octet long.

### 12.14 data.cache - Cache

**data.cache** [Module]
A cache works similarly as a dictionary, associating keys to values, but its entries may disappear according to the policy of the cache algorithm. This module defines a common protocol for cache datatypes, and also provides several typical cache implementations.

**Examples**
Let’s start from simple examples to get the idea.

Suppose you want to read given files and you want to cache the frequently read ones. The following code defines a cached version of file->string:

(use data.cache)
(use file.util)

(define file->string/cached
  (let1 file-cache (make-lru-cache 10 :comparator string-comparator)
   (^[path] (cache-through! file-cache path file->string))))

The procedure closes a variable file-cache, which is an LRU (least recently used) cache that associates string pathnames to the file contents. The actual logic is in cache-through!, which first consults the cache if it has an entry for the path. If the cache has the entry, its value (the file content) is returned. If not, it calls file->string with the path to fetch the file content, register it to the cache, and return it. The capacity of cache is set to 10 (the first
argument of \texttt{make-lru-cache}), so when the 11th file is read, the least recently used file will be purged from the cache.

The effect of cache isn’t very visible in the above example. You can insert some print stubs to see the cache is actually in action, as the following example. Try read various files using \texttt{file->string/cached}.

\begin{verbatim}
(define file->string/cached
  (let1 file-cache (make-lru-cache 10 :comparator string-comparator)
    (~ [path]
      (print #"file->string/cached called on ~path")
      (cache-through! file-cache path
        (~ [path]
          (print #"cache miss. fetching ~path")
          (file->string path))))))
\end{verbatim}

Caveat: A cache itself isn’t MT-safe. If you are using it in multithreaded programs, you have to wrap it with an atom (see Section 9.35.4 [Synchronization primitives], page 536):

\begin{verbatim}
(use data.cache)
(use file.util)
(use gauche.threads)

(define file->string/cached
  (let1 file-cache (atom (make-lru-cache 10 :comparator string-comparator))
    (~ [path]
      (atomic file-cache (cut cache-through! <> path file->string))))))
\end{verbatim}

Common properties of caches

A cache of any kind has a comparator and a storage. The comparator is used to compare keys; in the example above, we use \texttt{string-comparator} to compare string pathnames (see Section 6.2.4 [Basic comparators], page 116, for more about comparators).

The storage is a dictionary that maps keys to internal structures of the cache. By default, a hashtable is created automatically using the given comparator (or, if a comparator is omitted, using \texttt{default-comparator}). The comparator must have hash function.

Alternatively, you can give a pre-filled dictionary (copied from another instance of the same kind of cache) to start cache with some data already in it. Note that what the cache keeps in the dictionary totally depends on the cache algorithm, so you can’t just pass a random dictionary; it has to be created by the same kind of cache. If you pass in the storage, the comparator is taken from it.

Thus, the cache constructors uniformly take keyword arguments \texttt{comparator} and \texttt{storage}; you can specify either one, or omit both to use the defaults.

Predefined caches

For the \texttt{storage} and \texttt{comparator} keyword arguments, see above.

\begin{verbatim}
(make-fifo-cache capacity :key storage comparator)
  {data.cache} Creates and returns a FIFO (first-in, first-out) cache that can hold up to capacity entries. If the number of entries exceeds capacity, the oldest entry is removed.

(make-lru-cache capacity :key storage comparator)
  {data.cache} Creates and returns an LRU (least recently used) cache that can hold up to capacity entries. If the number of entries exceeds capacity, the least recently used entry is removed.
\end{verbatim}
make-ttl-cache timeout :key storage comparator timestamper
{data.cache} Creates and returns a TTL (time to live) cache with the timeout value timeout. Each entry is timestamped when it’s inserted, and it is removed when the current time passes timeout unit from the timestamp. The actual entry removal is done when the cache is accessed.

By default, the Unix system time (seconds from Epoch) is used as a timestamp, and timeout is in seconds. It may not be fine-grained enough if you add multiple entries in shorter intervals than seconds. You can customize it by giving a thunk to timestamper; the thunk is called to obtain a timestamp, which can be any monotonically increasing real number (it doesn’t need to be associated with physical time). If you give timestamper, the unit of timeout value should be the same as whatever timestamper returns.

make-ttlr-cache timeout :key storage comparator timestamper
{data.cache} A variation of TTL cache, but the entry’s timestamp is updated (refreshed) whenever the entry is read. Hence we call it TTL with refresh (TTLR). But you can also think it as a variation of LRU cache with timeout.

The unit of timeout, and the role of timestamper argument, are the same as make-ttl-cache.

Common operations of caches
The following APIs are for the users of a cache.

cache-lookup! cache key :optional default
{data.cache} Look for an entry with key in cache, and returns its value if it exists. If there’s no entry, the procedure returns default if it is provided, or throws an error otherwise.

Some types of cache algorithms update cache by this operation, hence the bang is in the name.

cache-through! cache key value-fn
{data.cache} Look for an entry with key in cache, and returns its value if it exists. If there’s no entry, a procedure value-fn is called with key as the argument, and its return value is inserted into cache and also returned.

cache-write! cache key value
{data.cache} This inserts association of key and value into cache. If there’s already an entry with key, it is overwritten. Otherwise a new entry is created.

The same effect can be achieved by calling cache-evict! then cache-through!, but cache algorithms may provide efficient way through this method.

cache-evict! cache key
{data.cache} Removes an entry with key from cache, if it exists.

cache-clear! cache
{data.cache} Removes all entries from cache.

Implementing a cache algorithm
Each cache algorithm must define a class inheriting <cache>, and implement the following two essential methods. The higher-level API calls them.

cache-check! cache key
{data.cache} Looks for an entry with key in cache. If it exists, returns a pair of key and the associated value. Otherwise, returns #f. It may update the cache, for example, the timestamp of the entry for being read.
cache-register!  

Add an entry with key and associated value into cache. This is called after key is confirmed not being in cache.

Additionally, the implementation should consider the following points.

- The initialize method must call next-method first, which sets up the comparator and storage slots. You should check if storage has pre-filled entries, and if so, set up other internal structures appropriately.
- The default methods of cache-evict! and cache-clear! only takes care of the storage of the cache. You should implement them if your auxiliary structure needs to be taken care of.
- The default method of cache-write! is just cache-evict! followed by cache-register!.

You may provide alternative method if you can do it more efficiently, which is often the case.

There are several procedures that help implementing cache subclasses:

**cache-comparator cache**  
Returns the comparator and the storage of the cache, respectively.

**cache-populate-queue! queue storage**  
This procedure walks storage (a dictionary that maps keys to (<n> . <value>)), and queues holds (<key> . <n>), <n> being a number (timestamp, counter, etc.) Here are some common operations work on this queue-and-dictionary scheme:

**cache-compact-queue! queue storage**  
The queue may contain multiple pairs with the same key. Sometimes the queue gets to have too many duplicated entries (e.g. the same entry is read repeatedly, and you push the read timestamp to the queue for every read). This procedure scans the queue and removes duplicated entries but the up-to-date one. After this operation, the length of the queue and the number of entries in the storage should match.

**cache-renumber-entries! queue storage**  
This procedure renumbers <n>s in the queue and the storage starting from 0, without changing their order, and returns the maximum <n>. The duplicated entries in the queue is removed as in cache-compact-queue!.

When you’re using monotonically increasing counter for <n> and you don’t want <n> to get too big (i.e. bignums), you can call this procedure occasionally to keep <n>’s in reasonable range.

### 12.15 data.heap - Heap

A heap is a data container that allows efficient retrieval of the minimum or maximum entry. Unlike a tree-map (see Section 6.14.2 [Treemaps], page 212), which always keeps all entries in order, a heap only cares the minimum or the maximum of the current set; the other entries are only partially ordered, and reordered when the minimum/maximum entry is removed. Hence it is more efficient than a treemap if all you need is minimum/maximum value. Besides binary heaps can store entries in packed, memory-efficient way.
<binary-heap>  
{data.heap} An implementation of a binary heap. Internally it uses min-max heap, so that you can find both minimum and maximum value in $O(1)$. Pushing a new value and popping the minimum/maximum value are both $O(\log n)$.

It also stores its values in a flat vector, a lot more compact than a general tree structure that needs a few pointers per node. By default it uses a sparse vector for the backing storage, allowing virtually unlimited capacity (see Section 12.24.1 [Sparse vectors], page 854). But you can use an ordinal vector or a uniform vector as a backing storage instead.

A binary heap isn’t MT-safe structure; you must put it in atom or use mutexes if multiple threads can access to it (see Section 9.35.4 [Synchronization primitives], page 536).

make-binary-heap :key comparator storage key  
{data.heap} Creates and returns a new binary heap. The comparator keyword argument specifies how to compare the entries. It must have comparison procedure or ordering predicate. The default is default-comparator. See Section 6.2.4 [Basic comparators], page 116, for the details of comparators.

The storage keyword argument gives alternative backing storage. It must be either a vector, a uniform vector, or an instance of a sparse vector (see Section 12.24.1 [Sparse vectors], page 854). The default is an instance of <sparse-vector>. If you pass a vector or a uniform vector, it determines the maximum number of elements the heap can hold. The heap won’t be extend the storage once it gets full.

The key keyword argument must be a procedure; it is applied on each entry before comparison. Using key procedure allows you to store auxiliary data other than the actual value to be compared. The following example shows the entries are compared by their car’s:

```lisp
(define *heap* (make-binary-heap :key car))
(binary-heap-push! *heap* (cons 1 'a))
(binary-heap-push! *heap* (cons 3 'b))
(binary-heap-push! *heap* (cons 1 'c))

(binary-heap-find-min *heap*)  ⇒  (1 . c)
(binary-heap-find-max *heap*)  ⇒  (3 . b)
```

build-binary-heap storage :key comparator key num-entries  
{data.heap} Create a heap from the data in storage, and returns it. (Sometimes this operation is called heapify.) This allows you to create a heap without allocating a new storage. The comparator and key arguments are the same as make-binary-heap.

Storage must be either a vector, a uniform vector, or an instance of a sparse vector. The storage is modified to satisfy the heap property, and will be used as the backing storage of the created heap. Since the storage will be owned by the heap, you shouldn’t modify the storage later.

The storage supposed to have keys from index 0 below num-entries. If num-entries is omitted or #f, entire vector or uniform vector, or up to sparse-vector-num-entries on the sparse vector, is heapified.

binary-heap-copy heap  
{data.heap} Copy the heap. The backing storage is also copied.

binary-heap-clear! heap  
{data.heap} Empty the heap.

binary-heap-num-entries heap  
{data.heap} Returns the current number of entries in the heap.
Chapter 12: Library modules - Utilities

binary-heap-empty? heap
{data.heap} Returns #t if the heap is empty, #f otherwise.

binary-heap-push! heap item
{data.heap} Insert item into the heap. This is O(log n) operation. If the heap is already full, an error is raised.

binary-heap-find-min heap :optional fallback
binary-heap-find-max heap :optional fallback
{data.heap} Returns the minimum and maximum entry of the heap, respectively. The heap will be unmodified. This is O(1) operation.

If the heap is empty, fallback is returned when it is provided, or an error is signaled.

binary-heap-pop-min! heap
binary-heap-pop-max! heap
{data.heap} Removes the minimum and maximum entry of the heap and returns it, respectively. O(log n) operation. If the heap is empty, an error is signaled.

The following procedures are not heap operations, but provided for the convenience.

binary-heap-swap-min! heap item
binary-heap-swap-max! heap item
{data.heap} These are operationally equivalent to the followings, respectively:

(begin0 (binary-heap-pop-min! heap)
 (binary-heap-push! heap item))

(begi0 (binary-heap-pop-max! heap)
 (binary-heap-push! heap item))

However, those procedures are slightly efficient, using heap property maintaining procedure only once per function call.

binary-heap-find pred heap :optional failure
{data.heap} Returns an item in the heap that satisfies pred. If there are more than one item that satisfy pred, any one of them can be returned. If no item satisfy pred, the thunk failure is called, whose default is (\[\] #f). This is O(n) operation.

Note: The argument order used to be (binary-heap-find heap pred) until 0.9.10. We changed it to align other *find procedures. The old argument order still work for the backward compatibility, but the new code should use the current order.

binary-heap-remove! heap pred
{data.heap} Remove all items in the heap that satisfy pred. This is O(n) operation.

binary-heap-delete! heap item
{data.heap} Delete all items in the heap that are equal to item, in terms of the heap’s comparator and key procedure. This is O(n) operation.

Note that the key procedure is applied to item as well before comparison.

12.16 data.ideque - Immutable deques

data.ideque

This module provides a functional double-ended queue (deque, pronounced as “deck”).

Almost all procedures in this module are now a part of R7RS large. See Section 10.3.10 [R7RS immutable deques], page 623, for description of the following procedures:
ideque ideque-unfold ideque-unfold-right
ideque-tabulate  ideque?  ideque-empty?
ideque-add-front  ideque-add-back
ideque-remove-front  ideque-remove-back
ideque-front  ideque-back
ideque-reverse  ideque=  ideque-ref
ideque-take  ideque-drop
ideque-take-right  ideque-drop-right
ideque-split-at  ideque-append  ideque-zip
ideque-map  ideque-for-each  ideque-for-each-right
ideque-fold  ideque-fold-right  ideque-append-map
ideque-filter  ideque-remove
ideque-find  ideque-find-right
ideque-take-while  ideque-take-while-right
ideque-drop-while  ideque-drop-while-right
ideque-span  ideque-break
ideque-any  ideque-every
ideque->list  list->ideque
ideque->generator  generator->ideque

make-ideque  n :optional init
{data.idealque} Creates an ideque of length n with all the elements being init. If init is omitted, #f is used.

This is provided just for the symmetry with other container data structures; it’s not in SRFI-134, and the portable code can use ideque-tabulate.

12.17 data.imap - Immutable map

data.imap

This module provides a immutable data structure with O(log n) access and update operations (here, update means to return a new structure with requested changes). The current implementation is based on the functional red-black tree.

Although lists and alists are useful for stack-like immutable operations, where you can add and remove items to the head of existing data without modifying them, they require O(n) access time and sometimes you need better one. The <imap> object provides O(log n) access, in exchange of O(log n) insertion and deletion.

<imap-meta>
{data.imap} Metaclass of <imap>.

<imap>
{data.imap} Immutable map class. An instance of <imap-meta>.
Inherits <ordered-dictionary>, conforms dictionary protocol except mutating operators (see Section 9.10 [Dictionary framework], page 428). As a sequence, you can access key-value pairs in increasing order of keys.

make-imap
{data.imap} Creates a new empty immutable map. Without arguments, default-comparator is used to compare keys. To give a specific comparator, use the second form; the comparator argument should have comparison procedure. For the details of comparators, see Section 6.2.4 [Basic comparators], page 116. The third form
creates a key comparator from a equality predicate key=? and less-than predicate key<?, both must accept two keys. This interface is consistent with \texttt{tree-map} (see Section 6.14.2 [Treemaps], page 212).

\begin{verbatim}
(alist->imap) \texttt{alist} \quad \textbf{[Function]}  
(alist->imap \texttt{alist comparator}) \quad \textbf{[Function]}  
(alist->imap \texttt{alist key=? key<?}) \quad \textbf{[Function]}  
\end{verbatim}

\{data.imap\} Creates and returns an immutable map containing key-value associations given by an assoc-list \texttt{alist}. This may be a bit more efficient than creating an empty map with \texttt{make-imap} and populates it with \texttt{imap-put} one by one.

The \textit{comparator} argument specifies how to compare the keys. It must have comparison procedure. If omitted, \texttt{default-comparator} is used. See Section 6.2.4 [Basic comparators], page 116, for the details.

The third form creates a key comparator from a equality predicate key=? and less-than predicate key<?, both must accept two keys.

\begin{verbatim}
(define m (alist->imap '((a . 1) (b . 2))))

(imap-get m 'a) \Rightarrow 1
(imap-get m 'b) \Rightarrow 2
\end{verbatim}

\begin{verbatim}
(tree-map->imap) \texttt{tree-map} \quad \textbf{[Function]}  
\end{verbatim}

\{data.imap\} Returns a new immutable map with the same content (and the same comparator) as \texttt{tree-map}.

\begin{verbatim}
imap? \texttt{obj} \quad \textbf{[Function]}  
imap-empty? \texttt{immap} \quad \textbf{[Function]}  
imap-exists? \texttt{immap key} \quad \textbf{[Function]}  
imap-get \texttt{immap key :optional default} \quad \textbf{[Function]}  
imap-put \texttt{immap key val} \quad \textbf{[Function]}  
imap-delete \texttt{immap key} \quad \textbf{[Function]}  
\end{verbatim}

\{data.imap\} Returns \#t if \textit{obj} is an immutable map, \#f otherwise.

\{data.imap\} Returns \#t if an immutable map \texttt{immap} is empty, \#f otherwise.

\{data.imap\} Returns \#t if \texttt{key} exists in an immutable map \texttt{immap}.

\{data.imap\} Returns the value associated with \texttt{key} in an immutable map \texttt{immap}. If \texttt{immap} doesn’t have \texttt{key}, \texttt{default} is returned when provided, otherwise an error is signalled.

\{data.imap\} Returns a new immutable map where association of \texttt{key} to \texttt{val} is added to (or replaced in) an immutable map \texttt{immap}. This operation is $O(\log n)$.

\begin{verbatim}
(define m1 (alist->imap '((a . 1) (b . 2))))

(define m2 (imap-put m1 'a 3))

(imap-get m2 'a) \Rightarrow 3
(imap-get m1 'a) \Rightarrow 1 \quad \text{; not affected}
\end{verbatim}

\{data.imap\} Returns a new immutable map where \texttt{key} is removed from \texttt{immap}. If \texttt{immap} doesn’t have \texttt{key}, returned map has the same content as \texttt{immap}.

\begin{verbatim}
(define m1 (alist->imap '((a . 1) (b . 2))))

(define m2 (imap-delete m1 'a))
\end{verbatim}
(imap-get m2 'a #f) ⇒ #f
(imap-get m1 'a) ⇒ 1 ; not affected

imap-min immap
imap-max immap
{data.imap} Returns a pair of key and value with the minimum or maximum key in immap, respectively. If immap is empty, #f is returned.

12.18 data.priority-map - Priority map

data.priority-map
Priority map is a dictionary that can map keys to values, while the entires are sorted by their values. (If the entires are sorted by keys, it’s a treemap–see Section 6.14.2 [Treemaps], page 212).
It is useful when you wanted a sorted sequence, with quick access by keys that are associated to each value.
Note: If what you need is just a priority queue, you can use data.heap (see Section 12.15 [Heap], page 831).

<priority-map>
{data.priority-map} Priority map class. It has no public slots. Instances of priority maps must be created by make-priority-map procedure instead of make method on the class.
Inherits <ordered-dictionary>, and implements dictionary protocol. Operations concerning associations of keys and values are done through dictionary generic functions (see Section 9.10.1 [Generic functions for dictionaries], page 428).
When iterated, it iterates increasing order of values.

make-priority-map :key key-comparator value-comparator
{data.priority-map} Creates and returns an empty priority map. It can take key-comparator, which must have a hash procedure and is used to hash keys, and value-comparator, which must have ordering predicate and is used to order values. When omitted, default-comparator is assumed.

priority-map-min pmap
priority-map-max pmap
{data.priority-map} Return a pair of a key and a value, where the value is smallest or largest in the priority map pmap, respectively.
If pmap is empty, #f is returned.
If pmap has more than one value that are equal to each other (w.r.t. value-comparator given to the constructor), either one of them is picked.

priority-map-min-all pmap
priority-map-max-all pmap
{data.priority-map} Return a pair of a list of keys and a value, where the value is smallest or largest in the priority map pmap, respectively. The keys includes all the entries that has the same value. The caller shouldn’t modify the returned key list.
If pmap is empty, #f is returned.

priority-map-pop-min! pmap
priority-map-pop-max! pmap
{data.priority-map} Remove one entry with the smallest or largest value, respectively, and returns a pair of the key and the value of removed entry.
If $pmap$ is empty, $\#f$ is returned.

If $pmap$ has more than one value that are equal to each other (w.r.t. value-comparator given to the constructor), either one of them is picked.

### 12.19 data.queue - Queue

#### data.queue  
[Module]

Provides a queue (FIFO). You can create a simple queue, which is lightweight but not thread-safe, or an MTqueue, a thread-safe queue. Basic queue operations work on both type of queues. When an mtqueue is passed to the procedures listed in this section, each operation is done in atomic way, unless otherwise noted.

There are also a set of procedures for mtqueues that can be used for thread synchronization; for example, you can let the consumer thread block if an mtqueue is empty, and/or the producer thread block if the number of items in the mtqueue reaches a specified limit. Using these procedures allows the program to use an mtqueue as a channel.

The simple queue API is a superset of SLIB’s queue implementation, which supports not only enqueue! (add item to the end of the sequence) and dequeue! (take item from the front of the sequence), but also queue-push! (add item to the front of the sequence), so that it can be used as a stack as well.

If you also want to take item from the end of the sequence in $O(1)$, you need a deque (double-ended queue). See Section 12.22 [Ring buffer], page 849, which works as an efficient (both speed and space) deque on top of vectors. Or you can use immutable deques provided by data.ideque (see Section 12.16 [Immutable deques], page 833).

See also scheme.list-queue (Section 10.3.16 [R7RS list queues], page 636), which defines a portable API for list-based queue.

#### <queue>  
[Class]

{data.queue} A class of simple queue.

length  
[Instance Variable of <queue>]

A read-only slot that returns the number of items in the queue.

#### <mtqueue>  
[Class]

{data.queue} A class of mtqueue. Inherits <queue>.

max-length  
[Instance Variable of <mtqueue>]

The upper bound of the number of items in the queue.

If this slot is zero, the queue cannot hold any items, but works as a synchronization device. A writer will block until a reader appears to take the item; a reader will block until a writer appears to give the item.

closed  
[Instance Variable of <mtqueue>]

A boolean flag, set to $\#f$ initially. If this is true, the queue no longer accepts a new data by enqueue! etc. This slot is read-only and can only be changed atomically by enqueue/wait!, queue-push/wait!, dequeue/wait! and queue-pop/wait!. This is useful when an mtqueue used as a channel is being shutdown.

make-queue  
{data.queue} Creates and returns an empty simple queue.

make-ntqueue :key max-length  
{data.queue} Creates and returns an empty mtqueue. When an integer is given to the keyword argument max-length, it is used to initialize the max-length slot.
queue? obj
{data.queue} Returns #t if obj is a queue (either a simple queue or an mtqueue).

mtqueue? obj
{data.queue} Returns #t if obj is an mtqueue.

queue-empty? queue
{data.queue} Returns #t if obj is an empty queue.

queue-length queue
{data.queue} Returns the number of the items in the queue.

mtqueue-max-length mtqueue
{data.queue} Returns the maximum number of items the mtqueue can hold. If the queue
doesn’t have a limit, #f is returned.

mtqueue-room mtqueue
{data.queue} Returns the number of elements the mtqueue can accept at this moment before
it hits its maximum length. For example, if the queue already has the maximum number of
elements, 0 is returned. If the queue doesn’t have the limit, +inf.0 is returned.

Note that even if this returns a non-zero finite value, subsequent enqueue! may throw an
error because of the queue being full. It’s because another thread may put an item to the
queue between this procedure call and enqueue!. To avoid this situation, use enqueue/wait!
to insert item to mtqueue with finite max-length.

mtqueue-num-waiting-readers mtqueue
{data.queue} Returns the number of threads waiting on the mtqueue to read at this moment.
The return value is always a nonnegative exact integer.

Note that the value might change between this procedure’s returning the value and your
checking it, if some other thread inserts an element into the queue. To use the value reliably,
you need another mutex to restrict putting items in the queue.

(define q (make-mtqueue))
(thread-start! (make-thread (^[[] (dequeue/wait! q))))

(mtqueue-num-waiting-readers q) ⇒ 1

(enqueue! q 'a)

(mtqueue-num-waiting-readers q) ⇒ 0

copy-queue queue
{data.queue} Returns a copy of the queue.

enqueue! queue obj :optional more-objs ... 
{data.queue} Add obj to the end of queue. You may give more than one object, and each
of them are enqueued in order.

If queue is an mtqueue, all the objects are enqueued atomically: no other objects from other
threads can be inserted between the objects given to a single enqueue! call. Besides, if the
value of its max-length slot has a positive finite value, and adding objs makes the number
of elements in queue exceeds max-length, an error is signaled and queue won’t be modified.
(If max-length is zero, this procedure always fail. Use enqueue/wait! below.)

If queue is an mtqueue and it is closed, no change is made to it and an error is thrown.
Chapter 12: Library modules - Utilities

queue-push! queue obj :optional more-objs ...
{data.queue} Add obj in front of queue. You may give more than one object, and each of them are pushed in order.

Like enqueue!, when queue is an mtqueue, all objects are added atomically, and the value of max-length slot is checked. See enqueue! above for the details.

enqueue-unique! queue eq-proc obj :optional more-objs ...
queue-push-unique! queue eq-proc obj :optional more-objs ...
{data.queue} Like enqueue! and queue-push!, respectively, except that these don’t modify queue if it already contains obj (elements are compared by two-argument procedure eq-proc).

When queue is an mtqueue, all objects are added atomically, and the value of max-length slot is checked. See enqueue! above for the details.

dequeue! queue :optional fallback
dequeue-pop! queue :optional fallback
{data.queue} Take one object from the front of the queue queue and returns it. Both function works the same, but dequeue-pop! may be used to emphasize it works with queue-push!.

If queue is empty, fallback is returned if given, otherwise an error is signaled.

If queue is an mtqueue and its max-length is zero, the queue is always empty. Use dequeue/wait! to use such a queue as an synchronization device.

dequeue-all! queue
{data.queue} Returns the whole content of the queue by a list, with emptying queue. If queue is already empty, returns an empty list. See also queue->list below.

queue-front queue :optional fallback
queue-rear queue :optional fallback
{data.queue} Peek the head or the tail of the queue and returns the object, respectively. The queue itself is not modified. If queue is empty, fallback is returned if it is given, otherwise an error is signaled.

list->queue list :optional class :rest initargs
{data.queue} Returns a new queue whose content is the elements in list, in the given order.

By default the created queue is a simple queue, but you can create mtqueue or instances of other subclasses of <queue> by giving the class to the optional class arguments. The optional initargs arguments are passed to the constructor of class.

queue->list queue
{data.queue} Returns a list whose content is the items in the queue in order. Unlike dequeue-all!, the content of queue remains intact.

In Gauche, queue->list copies the content of the queue to a freshly allocated list, while dequeue-all! doesn’t copy but directly returns the queue’s internal list. There are some Scheme systems that has queue->list but doesn’t guarantee the content is copied, so if you’re planning to share the code among these implementations, it’s better not to rely on the fact that queue->list copies the content.

queue-internal-list queue
{data.queue} Like queue->list, returns a list whose content is the items in the queue in order, but the returned list may share the internal storage of queue. The returned list can be modified by subsequent operations of queue, and any modification on the list can make queue inconsistent.

Because of this danger, we don’t allow <mtqueue> to be passed to this procedure; it would signal an error if you do so.
If you just want to extract the accumulated result in queue without copying, consider dequeue-all!, which is safe because it atomically resets the queue. Use this procedure only when you absolutely need to access the contents of the queue without taking them out.

**find-in-queue** pred queue
{data.queue} Returns the first item in queue that satisfies a predicate pred. The order of arguments follows find (see Section 6.6.7 [Other list procedures], page 151).

**any-in-queue** pred queue
{data.queue} Like any in SRFI-1, apply pred on each item in queue until it evaluates true, and returns that true value (doesn’t necessarily be #t). If no items in the queue satisfies pred, #f is returned.

**every-in-queue** pred queue
{data.queue} Like every in SRFI-1, apply pred on each item in queue. If pred returns #f, stops iteration and returns #f immediately. Otherwise, returns the result of the application of pred on the last item of the queue. If the queue is empty, #t is returned.

**remove-from-queue!** pred queue
{data.queue} Removes all items in the queue that satisfies pred. Returns #t if any item is removed. Otherwise returns #f. The order of arguments follows remove in scheme.list (see Section 10.3.1 [R7RS lists], page 593).

Note on portability: Scheme48 has delete-from-queue!, which takes object to remove rather than predicate, and also takes arguments in reversed order (i.e. queue comes first). Avoid conflicting with that I intentionally left out delete-from-queue!: it’s easy to write one in either Scheme48 compatible way or consistent to SRFI-1 argument order.

**enqueue/wait!** mtqueue obj :optional timeout timeout-val close

**queue-push/wait!** mtqueue obj :optional timeout timeout-val close

**dequeue(wait!** mtqueue :optional timeout timeout-val close

**queue-pop/wait!** mtqueue :optional timeout timeout-val close
{data.queue} These synchronizing variants allows an mtqueue to be used as a “channel”, which communicates producer thread(s) and consumer thread(s).

The caller thread would block if the mtqueue has reached its maximum length (for enqueue/wait! and queue-push/wait!), or the mtqueue is empty (for dequeue(wait! and queue-pop/wait!). The blocked caller thread is unblocked either when the blocking condition is resolved, or the timeout condition is met.

The optional timeout argument specifies the timeout condition. If it is #f, those procedures wait indefinitely. If it is a real number, they wait at least the given number of seconds. If it is a <time> object (see Section 6.24.9 [Time], page 316), they wait until the absolute point of time the argument specifies.

In case the call is blocked then timed out, the value of timeout-val is returned, which defaults to #f.

When enqueue/wait! and queue-push/wait! succeeds without hitting timeout, they return #t.

If mtqueue is already closed, enqueue/wait! and queue-push/wait! raise an error, without modifying the queue. The check and queue insertion is done atomically, to eliminate the possibility that other thread tries to enqueue between the check and insertion. You can use dequeue/wait! and queue-pop/wait! on a closed mtqueue.

The last optional argument, close, closes the queue if it is given and true. The close operation is done atomically, and if you’re calling enqueue/wait! or queue-push/wait!, obj is guaranteed to be the last item put in the queue. It effectively “shut down” the channel.
12.20 data.random - Random data generators

This module defines a set of generators and generator makers that yield random data of specific type and distribution.

A naming convention: Procedures that takes parameters and returns a generator is suffixed by $ (e.g. integer$). Procedures that are generators themselves are not (e.g. fixnums). Procedures that are combinators, that is, the ones that take one or more generators and returns a generator, generally ends with a preposition (e.g. list-of).

Global state

All the generators in this module shares a global random state. The random seed is initialized by a fixed value when the module is loaded. You can get and set the random seed by the following procedure.

(random-data-seed) ⇒ 1

Note: This procedure doesn’t have parameter interface (alter the global value by giving the new value as an argument), since it doesn’t work like a parameter (see Section 6.16.1 [Parameters], page 233). You can get the random seed value, but you can’t get the current random state itself—if you restore the random seed value again, the internal state is reset, instead of restoring the state at the time you called random-data-seed.

If you want to use different random state temporarily, and ensure to restore original state afterwards, use with-random-data-seed below.

Generators of primitive data types

Those generators generate uniformly distributed data.

In the following examples, we use generator->list to show some concrete data from the generators. It is provided in gauche.generator module. See Section 9.12 [Generators], page 436, for more utilities work on generators.

(integers$ size :optional (start 0))
(integers-between$ lower-bound upper-bound)
second one, \texttt{integers-between\$}, creates a generator that generates integers between lower-bound and upper-bound (both inclusive) uniformly.

\begin{verbatim}
;; A dice roller
(define dice (integers\$ 6 1))

;; Roll the dice 10 times
(generator->list dice 10)
⇒ (6 6 2 4 2 5 5 1 2 2)
\end{verbatim}

\begin{description}
\item fixnums
\item int8s
\item uint8s
\item int16s
\item uint16s
\item int32s
\item uint32s
\item int64s
\item uint64s
\end{description}

\texttt{\{data.random\}} Uniform integer generators. Generate integers in fixnum range, and 8/16/32/64bit signed and unsigned integers, respectively.

\begin{verbatim}
(generator->list int8s 10)
\end{verbatim}

\begin{description}
\item booleans
\end{description}

\texttt{\{data.random\}} Generates boolean values (\texttt{#f} and \texttt{#t}) in equal probability.

\begin{verbatim}
(generator->list booleans 10)
⇒ (#f #f #t #f #f #t #f #f #f #f)
\end{verbatim}

\begin{description}
\item chars\$: \texttt{optional char-set}
\end{description}

\texttt{\{data.random\}} Creates a generator that generates characters in \texttt{char-set} uniformly. The default \texttt{char-set} is \texttt{[#A-Za-z0-9]}.

\begin{verbatim}
(define alphanumeric-chars (chars\$))
(generator->list alphanumeric-chars 10)
⇒ (#\f #\m #\3 #\S #\z #\m #\x #\S #\l #\y)
\end{verbatim}

\begin{description}
\item reals\$: \texttt{optional size start}
\item reals-between\$: \texttt{lower-bound upper-bound}
\end{description}

\texttt{\{data.random\}} Create a generator that generates real numbers uniformly with given range. The first procedure, \texttt{reals\$}, returns reals between start and start+size, inclusively. The default of size is 1.0 and start is 0.0. The second procedure, \texttt{reals-between\$}, returns reals between lower-bound and upper-bound, inclusively.

\begin{verbatim}
(define uniform-100 (reals\$ 100))
(generator->list uniform-100 10)
⇒ (81.67965004942268 81.84927577572596 53.02443813660833)
\end{verbatim}

Note that a generator from \texttt{reals\$} can generate the upper-bound value start+size, as opposed to \texttt{integers\$}. If you need to exclude the bound value, just discard the bound value; \texttt{gfilter} may come handy.

\begin{verbatim}
(define generate-from-0-below-1
 (gfilter (^r (not (= r 1.0))) (reals\$ 1.0 0.0)))
\end{verbatim}
samples$  collection  [Function]
{data.random} Creates a generator that returns randomly chosen item in collection at a
time.

Do not confuse this with samples-from below, which is to combine multiple generators for
sampling.

(define coin-toss (samples$ '(head tail)))

(generator->list coin-toss 5)
⇒ (head tail tail head tail)

regular-string$  regexp  [Function]
{data.random} Creates an infinite generator that generates random strings each of which
matches the given regexp. The regexp shouldn’t include conditional patterns and looka-
head/behind assertions.

Note: It is hard to define how the distribution of the generated strings should look like. For
now, we build an NFA from regexp and put the same probability when there are multiple
choices, but that may not be really useful for typical use cases (e.g. generate test data).

Please assume the current implementation strategy a provisional one.

Nonuniform distributions

reals-normal$  :optional mean deviation  [Function]
{data.random} Creates a generator that yields real numbers from normal distribution with
mean and deviation. The default of mean is 0.0 and deviation is 1.0.

reals-exponential$  mean  [Function]
{data.random} Creates a generator that yields real numbers from exponential distribution
with mean.

integers-geometric$  p  [Function]
{data.random} Creates a generator that yields integers from geometric distribution with
success probability p (0 <= p <= 1). The mean is 1/p and variance is (1-p)/p^2.

integers-poisson$  L  [Function]
{data.random} Creates a generator that yields integers from poisson distribution with mean
L, variance L.

Aggregate data generators

samples-from  generators  [Function]
{data.random} Takes a finite sequence of generators (sequence in the sense of
gauche.sequence), and returns a generator. Every time the resulting generator is called, it
picks one of the input generators in equal probability, then calls it to get a value.

(define g (samples-from (list uint8s (chars$ #*[a-z]*))))

(generator->list g 10)
⇒ (207 107 #\m #\f 199 #\o #\b 57 #\j #\e)

NB: To create a generator that samples from a fixed collection of items, use samples$ de-
dcribed above.

weighted-samples-from  weight&gens  [Function]
{data.random} The argument is a list of pairs of a nonnegative real number and a generator.
The real number determines the weight, or the relative probability that the generator is
chosen. The sum of weight doesn’t need to be 1.0.
The following example chooses the uint8 generator four times frequently than the character generator.

\[
\text{(define g (weighted-samples-from '(((4.0 . ,uint8s) (1.0 . ,(chars$))))}}
\]

\[(\text{generator->list g 10)}
\Rightarrow (195 97 \#j \#W \#5 72 49 143 19 164)
\]

\text{pairs-of car-gen cdr-gen [Function]}
\{
\text{data.random} \text{ Returns a generator that yields pairs, whose car is generated from car-gen and whose cdr is generated from cdr-gen.}
\}

\[
\text{(define g (pairs-of int8s booleans))}
\]

\[(\text{generator->list g 10)}
\Rightarrow ((113 . #t) (101 . #f) (12 . #t) (68 . #f) (-55 . #f))
\]

\text{tuples-of gen ... [Function]}
\{
\text{data.random} \text{ Returns a generator that yields lists, whose i-th element is generated from the i-th argument.}
\}

\[
\text{(define g (tuples-of int8s booleans (char$)))}
\]

\[(\text{generator->list g 3)}
\Rightarrow ((-43 #f \#8) (53 #f \#1) (-114 #f \#i))
\]

\text{permutations-of seq [Function]}
\{
\text{data.random} \text{ Returns a generator that yields a random permutations of seq.}
\}

The type of seq should be a sequence with a builder (see Section 9.31 [Sequence framework], page 510). The type of generated objects will be the same as seq.

\[
\text{(generator->list (permutations-of '(1 2 3)) 3)}
\Rightarrow ((1 2 3) (2 3 1) (3 2 1))
\]

\[
\text{(generator->list (permutations-of "abc") 3)}
\Rightarrow ("cba" "cba" "cab")
\]

\text{combinations-of size seq [Function]}
\{
\text{data.random} \text{ Returns a generator that yields a sequence of size elements randomly picked from seq.}
\}

The type of seq should be a sequence with a builder (see Section 9.31 [Sequence framework], page 510). The type of generated objects will be the same as seq.

\[
\text{(generator->list (combinations-of 2 '(a b c)) 5)}
\Rightarrow ((a c) (a b) (a c) (b a) (a c))
\]

\[
\text{(generator->list (combinations-of 2 '#(a b c)) 5)}
\Rightarrow (\#(a c) \#(b c) \#(c b) \#(b a) \#(b c))
\]

The following procedures takes optional sizer argument, which can be either a nonnegative integer or a generator of nonnegative integers. The value of the sizer determines the length of the result data.

Unlike most of Gauche procedures, sizer argument comes before the last argument when it is not omitted. We couldn’t resist the temptation to write something like (lists-of 3 booleans).

If sizer is omitted, the default value is taken from the parameter default-sizer. The default of default-sizer is (integers-poisson$ 4).
lists-of item-gen [Function]
lists-of sizer item-gen [Function]
vectors-of item-gen [Function]
vectors-of sizer item-gen [Function]
strings-of [Function]
strings-of item-gen [Function]
strings-of sizer item-gen [Function]

{data.random} Creates a generator that generates lists, vectors or strings of values from item-gen, respectively. The size of each datum is determined by sizer.

You can also omit item-gen for strings-of. In that case, a generator created by (chars$) is used.

(generator->list (lists-of 3 uint8s) 4)
⇒ ((254 46 0) (77 158 46) (1 134 156) (74 5 110))

;; using the default sizer
(generator->list (lists-of uint8s) 4)
⇒ ((93 249) (131 97) (98 206 144 247 241) (126 156 31))

;; using a generator for the sizer
(generator->list (strings-of (integers$ 8) (chars$)) 5)
⇒ ("dTJYVhu" "F" "PXkC" "w" "")

sequences-of class item-gen [Function]
sequences-of class sizer item-gen [Function]

{data.random} Creates a generator that yields sequences of class class, whose items are generated by item-gen. The size of each sequence is determined by sizer, or the value of default-sizer if omitted; the sizer can be a nonnegative integer, or a generator that yields nonnegative integers.

The class class must be a subclass of <sequence> and implement the builder interface.

(generator->list (sequences-of <u8vector> 4 uint8s) 3)
⇒ (#u8(95 203 243 46) #u8(187 199 153 152) #u8(39 114 39 25))

default-sizer [Parameter]
{data.random} The sizer used by lists-of, vectors-of and strings-of when sizer argument is omitted.

The value must be either a nonnegative integer, or a generator of nonnegative integers.

12.21 data.range - Range

data.range [Module]
A range object is an immutable sequence with O(1) indexed access to the elements, and each element may be computed procedurally. For example, a range of integers between 0 and N can be trivially realized as a range, where i-th element is simply computed by identity. It is a lot more space-efficient than actually generating a sequence containing every number.

It also allows certain operations efficient, such as taking subsequence or appending sequences.

A range object can be used with :range qualifier in the SRFI-42 eager comprehension (see Section 11.11 [Eager comprehensions], page 711).

A portable range object interface is defined in srfi.196. We implement the range object as a <sequence>, so that all sequence framework interface can be used on the ranges (see Section 9.31 [Sequence framework], page 510). We also provide some additional procedures that are not in SRFI-196.
Classes

<range> [Class]
Range object class. Internally we use several subclasses according to the nature of the range, but externally all ranges can be treated as an instance of <range>.

<range-meta> [Class]
The metaclass of <range> class.

Constructors

range length indexer [Function]
[SRFI-196]{data.range} Creates and returns a range of length length, whose i-th element (0 <= i < length)) is determined by a procedure indexer, which takes i as the sole argument and returns the corresponding element.

Note that indexer must run in O(1) time, and must be referentially transparent. The implementation may “expand” the range, that is, computes the element values into a flat vector internally.

        (range->list (range 5 (^i (+ i 10))))
⇒ (10 11 12 13 14)

numeric-range start end :optional step [Function]
[SRFI-196]{data.range} Creates and returns a range of integers starting from start (inclusive) and ending below end (exclusive), increasing with step. The default value of step is 1.

        (range->list (numeric-range 2 6))
⇒ (2 3 4 5)

        (range->list (numeric-range 0 5 2/3))
⇒ (0 2/3 4/3 2 8/3 10/3 4 14/3)

iota-range length :optional start step [Function]
[SRFI-196]{data.range} Creates and returns a range of integers. Total length of the range is length. The range starts from start (default 0), and increased with step (default 1).

        (range->list (iota-range 5))
⇒ (0 1 2 3 4)

        (range->list (iota-range 7 1 -1/7))
⇒ (1 6/7 5/7 4/7 3/7 2/7 1/7)

vector-range vec :optional start end [Function]
[SRFI-196+]{data.range} Returns a range over the given vector vec. The vector is kept in the range, so you shouldn’t mutate vec.

The optional start and end arguments limits the range of the vector to be used. They are Gauche’s extension and not in SRFI-196.

        (range->list (vector-range '(a b c)))
⇒ (a b c)

        (range->list (vector-range '(a b c d e) 1 4))
⇒ (b c d)
uvector-range uvec :optional start end  [Function]
{data.range} Returns a range over the given uniform vector uvec. The uniform vector is kept in the range, so you shouldn’t mutate uvec.

The optional start and end arguments limits the range of the uniform vector to be used.

bitvector/bool-range bvec :optional start end  [Function]
bitvector/int-range bvec :optional start end  [Function]
{data.range} Returns a range over the given bitvector bvec. Bitvector/bool-range regards bvec as a boolean vector, while bitvector/int-range regards it as a vector of 0 and 1. See Section 6.13.3 [Bitvectors], page 203, for the details of bitvectors. Note that bvec is kept in the range, so you shouldn’t mutate uvec.

The optional start and end arguments limits the range of the uniform vector to be used.

string-range str :optional start end  [Function]
{SRFI-196} Returns a range over each character in the given string str. The string is kept in the range, so you shouldn’t mutate str.

The optional start and end arguments limits the range of the vector to be used. They are Gauche’s extension and not in SRFI-196.

(range->list (string-range "abc"))
⇒ (#\a #\b #\c)

(range->list (string-range "abcd" 1 4))
⇒ (#\b #\c #\d)

range-append range . . .  [Function]
{SRFI-196} Returns a new range that walks over concatenation of the given ranges.

range-reverse range :optional start end  [Function]
{SRFI-196} Returns a new range that walks over the elements of range, but in reverse order.

The optional start and end arguments limits the range of the vector to be used. They are Gauche’s extension and not in SRFI-196.

(range->list (range-reverse (string-range "abc")))
⇒ (#\c #\b #\a)

(range->list (range-reverse (string-range "abcdef" 1 4)))
⇒ (#\d #\c #\b)

Predicates

range? obj  [Function]
{SRFI-196} Returns true iff obj is a range.

range=? elt= range . . .  [Function]
{SRFI-196} Returns true iff all ranges have the same length, and any pair of corresponding elements in the given ranges are equal in terms of elt= predicate.

As edge cases, when zero or one range is given, #t is returned.

(range=? eqv? (numeric-range 0 5)
 (iota-range 5)
 (vector-range '\#(0 1 2 3 4)))
⇒ #t
Accessors

**range-length range**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\) Returns the length of *range*.

**range-ref range i :optional fallback**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\) Returns the *i*-th element of *range*. The index *i* must be an exact integer.

If *i* is negative, or greater than or equal to the length of *range*, *fallback* is returned if given, or an error is signaled. The *fallback* argument is Gauche’s extension and not in SRFI-196.

**range-first range :optional fallback**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\) Returns the first element of *range*.

If the range is empty, *fallback* is returned if given, or an error is signaled. The *fallback* argument is Gauche’s extension and not in SRFI-196.

**range-last range :optional fallback**

[Function]  
\([\text{SRFI-196+}}\{\text{data.range}\}]\) Returns the last element of *range*.

If the range is empty, *fallback* is returned if given, or an error is signaled. The *fallback* argument is Gauche’s extension and not in SRFI-196.

Iteration

**range-split-at range k**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\) Returns two ranges, the first one with elements before *k*-th elements of *range*, and the second one with *k*-th elements and after of *range*.

**subrange range start end**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\) Returns a new range that contains *start*-th (inclusive) to *end*-th (exclusive) elements of *range*.

**range-segment range len**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-take range count**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-take-right range count**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-drop range count**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-drop-range range count**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-count pred range range2 . . .**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-any pred range range2 . . .**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-every pred range range2 . . .**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-map proc range range2 . . .**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-map->list proc range range2 . . .**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)

**range-map->vector proc range range2 . . .**

[Function]  
\([\text{SRFI-196}}\{\text{data.range}\}]\)
range-for-each  pred range range2 ...  [Function]  [SRFI-196]{data.range}
range-filter-map  pred range range2 ...  [Function]  [SRFI-196]{data.range}
range-filter-map->list  pred range range2 ...  [Function]  [SRFI-196]{data.range}
range-filter  pred range  [Function]  [SRFI-196]{data.range}
range-filter->list  pred range  [Function]  [SRFI-196]{data.range}
range-remove  pred range  [Function]  [SRFI-196]{data.range}
range-remove->list  pred range  [Function]  [SRFI-196]{data.range}
range-fold  kons knil range range2 ...  [Function]  [SRFI-196]{data.range}
range-fold-right  kons knil range range2 ...  [Function]  [SRFI-196]{data.range}
range-index  pred range range2 ...  [Function]  [SRFI-196]{data.range}
range-index-right  pred range range2 ...  [Function]  [SRFI-196]{data.range}
range-take-while  pred range  [Function]  [SRFI-196]{data.range}
range-take-while-right  pred range  [Function]  [SRFI-196]{data.range}
range-drop-while  pred range  [Function]  [SRFI-196]{data.range}
range-drop-while-right  pred range  [Function]  [SRFI-196]{data.range}

Conversion
range->list  range  [Function]  [SRFI-196]{data.range}
range->vector  range  [Function]  [SRFI-196]{data.range}
range->string  range  [Function]  [SRFI-196]{data.range}
vector->range  vec :optional start end  [Function]  [SRFI-196]{data.range}
range->generator  range :optional start end  [Function]  [SRFI-196]{data.range}

12.22 data.ring-buffer - Ring buffer

data.ring-buffer  [Module]
A ring buffer is an array with two fill pointers; in a typical usage, a producer adds new data to one end while a consumer removes data from the other end; if fill pointer reaches at the end of the array, it wraps around to the beginning, hence the name.

The ring buffer of this module allows adding and removing elements from both ends, hence functionally it is a double-ended queue, or deque. It also allows O(1) indexed access to the contents, and customized handling for the case when the buffer gets full.

You can use an ordinary vector or a uniform vector as the backing storage of a ring buffer.
**make-ring-buffer** :optional initial-storage :key overflow-handler

*initial-head-index initial-tail-index*

{data.ring-buffer} Creates a ring buffer. By default, a fresh vector is allocated for the backing storage. You can pass a vector or a uvector to *initial-storage* to be used instead. The passed storage must be mutable, and will be modified by the ring buffer; the caller shouldn’t modify it, nor make assumption about its content.

By default, the storage you pass as *initial-storage* is assumed to be an empty buffer. You can also pass a pre-filled storage, by specifying valid range of existing data with *initial-head-index* and *initial-tail-index* arguments (both are defaulted to 0). For example, the following code returns a ring buffer of initial capacity 8, with the first 4 items are already filled.

```
(make-ring-buffer (vector 'a 'b 'c 'd #f #f #f #f) :initial-tail-index 4)
```

The *overflow-handler* keyword argument specifies what to do when a new element is about to be added to the full buffer. It must be a procedure, or a symbol *error* or *overwrite*.

If it is a procedure, it will be called with a ring buffer and a backing storage (vector or uvector) when it is filled. The procedure must either (1) allocate and return a larger vector/uvector of the same type of the passed backing storage, (2) return a symbol *error*, or (3) return a symbol *overwrite*. If it returns a vector/uvector, it will be used as the new backing storage. The returned vector doesn’t need to be initialized; the ring buffer routine takes care of copying the necessary data. If it returns *error*, an error (“buffer is full”) is thrown. If it returns *overwrite*, the new element overwrites the existing element (as if one element from the other end is popped and discarded.)

Passing a symbol *error* or *overwrite* to *overflow-handler* is a shorthand of passing a procedure that unconditionally returns *error* or *overwrite*, respectively.

The default behavior on overflow is to double the size of backing storage. You can use **make-overflow-doubler** below to create the customized overflow handler easily.

**make-overflow-doubler** :key max-increase max-capacity

{data.ring-buffer} Returns a procedure suitable to be passed to the *overflow-handler* keyword argument of **make-ring-buffer**.

The returned procedure takes a ring buffer and its backing storage, and behaves as follows.

- If the size of current backing storage is equal to or greater than *max-capacity*, returns *error*.
- Otherwise, if the size of current backing storage is equal to or greater than *max-increase*, allocates a vector/uvector of the same type of the current backing storage, with the size (+ *max-increase* size-of-current-storage).
- Otherwise, allocates a vector/uvector of the same type of the current backing storage with the size (* 2 size-of-current-storage).

The default value of *max-increase* and *max-capacity* is *+inf.0*.

**ring-buffer-empty?** rb

{data.ring-buffer} Returns #t if the ring buffer rb is empty, #f if not.

**ring-buffer-full?** rb

{data.ring-buffer} Returns #t if the ring buffer rb is full, #f if not.

**ring-buffer-num-entries** rb

{data.ring-buffer} Returns the number of current elements in the ring buffer rb.

**ring-buffer-capacity** rb

{data.ring-buffer} Returns the size of the current backing storage of the ring buffer rb.
**Chapter 12: Library modules - Utilities 851**

**Function**

`ring-buffer-front rb`  
Returns the element in the front or back of the ring buffer `rb`, respectively. If the buffer is empty, an error is signaled.

`ring-buffer-back rb`  
Returns the element in the front or back of the ring buffer `rb`, respectively. If the buffer is empty, an error is signaled.

**Function**

`ring-buffer-add-front! rb elt`  
Add an element to the front or back of the ring buffer `rb`, respectively. If `rb` is full, the behavior is determined by the buffer’s overflow handler, as described in `make-ring-buffer`.

`ring-buffer-add-back! rb elt`  
Add an element to the front or back of the ring buffer `rb`, respectively. If `rb` is full, the behavior is determined by the buffer’s overflow handler, as described in `make-ring-buffer`.

**Function**

`ring-buffer-remove-front! rb`  
Remove an element from the front or back of the ring buffer `rb`, and returns the removed element, respectively. If the buffer is empty, an error is signaled.

`ring-buffer-remove-back! rb`  
Remove an element from the front or back of the ring buffer `rb`, and returns the removed element, respectively. If the buffer is empty, an error is signaled.

**Function**

`ring-buffer-ref rb index :optional fallback`  
Returns `index`-th element in the ring buffer `rb`. The elements are counted from the front; thus, if a new element is added to the front, the indexes of existing elements will shift.

If the index out of bounds of the existing content, `fallback` will be returned; if `fallback` is not provided, an error is signaled.

**Function**

`ring-buffer-set! rb index value`  
Sets `index`-th element of the ring buffer `rb` to `value`. The elements are counted from the front; thus, if a new element is added to the front, the indexes of existing elements will shift.

An error is signaled if the index is out of bounds.

**Function**

`ring-buffer->flat-vector rb :optional start end`  
Returns the current valid content of the ring buffer `rb` as a fresh flat vector of the same type as `rb`’s storage. If the optional `start/end` indexes are given, the content between those indexes are taken.

```
(define rb (make-ring-buffer (make-vector 4)))
(ring-buffer->flat-vector rb) ⇒ #()  
(ring-buffer-add-back! rb 'a)
(ring-buffer-add-back! rb 'b)
(ring-buffer-add-front! rb 'z)
(ring-buffer-add-front! rb 'y)
(ring-buffer->flat-vector rb) ⇒ #(y z a b)
(ring-buffer->flat-vector rb 1 3) ⇒ #(z a)
```

**12.23 data.skew-list - Skew binary random-access lists**

**Module**

This module implements skew binary random-access list (we call it skew-list for short). It’s an immutable data structure that has properties of both list and vector; constant time to take the first element (car) and append an element in front of existing one (cons), O(log n) to take the rest of the elements (cdr), and O(log n) for indexed access, where n is the number of elements.
A skew-list is always ‘proper’; that is, it is either an empty skew-list (`skew-list-null`), or an object appended in front of a skew-list.

<ske-list> [Class]
{data.skew-list} The class for skew-lists.
It inherits `<sequence>` and implements the sequence protocol (see Section 9.31 [Sequence framework], page 510).

skew-list? obj [Function]
{data.skew-list} Returns #t iff obj is a skew-list.

skew-list-empty? sl [Function]
{data.skew-list} The argument must be a skew list. Returns #t iff sl is an empty skew list.

skew-list-null [Variable]
{data.skew-list} An empty skew list.

skew-list-cons obj sl [Function]
{data.skew-list} Returns a new skew-list which has obj prepended to a skew-list sl. O(1) operation.

skew-list-car sl [Function]
{data.skew-list} Returns the first element of a skew-list sl. O(1) operation. An error is raised when sl is empty.

skew-list-cdr sl [Function]
{data.skew-list} Returns a new skew-list that contains elements in sl but the first one. O(log n) operation. An error is raised when sl is empty.

skew-list-ref sl k :optional fallback [Function]
{data.skew-list} Returns the k-th element of a skew-list sl. O(log n) operation. If k points past the range of sl, fallback is returned if it is given, or an error is signaled.
Sicne a skew-list is also a sequence, you can use generic ref as well (see Section 9.31 [Sequence framework], page 510).

skew-list-set sl k obj [Function]
{data.skew-list} Returns a new skew-list which is the same as sl except the k-th element is replaced with obj. The argument sl remains intact. O(log n) operation.
It is an error if k points beyond the last element of sl.

skew-list-length sl [Function]
{data.skew-list} Returns an integer length of a skew-list sl. O(log n) operation.
Sicne a skew-list is also a sequence, you can use generic size-of as well (see Section 9.31 [Sequence framework], page 510).

skew-list-length<=? sl n [Function]
{data.skew-list} Returns #t iff the length of a skew-list sl is less than or equal to n. It is more efficient than calculating the total length and compares with n.

list->skew-list lis [Function]
{data.skew-list} Returns a new skew-list which contains elements in lis, with the same order. It is an error if lis is not a proper list.
Sicne a skew-list is also a sequence, you can use generic coerce-to as well (see Section 9.31 [Sequence framework], page 510).
list*->skew-list lis
{data.skew-list} The argument lis may be a dotted list (but can’t be circular). Returns two values: a new skew-list which contains elements in lis except the last cdr of it, and the last cdr of lis.

skew-list->list sl
{data.skew-list} Returns a new list that contains all the elements in a skew-list sl, with the same order.

Sicne a skew-list is also a sequence, you can use generic coerce-to as well (see Section 9.31 [Sequence framework], page 510.

skew-list->generator sl
{data.skew-list} Returns a new generator that traverses a skew-list sl.

skew-list->lseq sl
{data.skew-list} Converts a skew-list sl to a lazy sequence.

skew-list-take sl k
skew-list-drop sl k
{data.skew-list} Returns a new skew-list of the first k elements and the elements except the first k elements, respectively.

skew-list-split-at sl k
{data.skew-list} Returns two values, the result of (skew-list-take sl k) and (skew-list-drop sl k), but more efficiently.

skew-list-append sl sl2 . . .
{data.skew-list} Returns a skew-list which is a concatenation of all the given skew-lists.

skew-list-fold sl kons knil
{data.skew-list} Like fold on a list.

Sicne a skew-list is also a sequence, you can use generic fold as well (see Section 9.31 [Sequence framework], page 510.

skew-list-map sl proc
{data.skew-list} Returns a skew list, each element of which is the result of applying proc on the element of sl. The order of application of proc is not specified.

NB: If you want to map over multiple skew-lists, you can use generic map (see Section 9.31 [Sequence framework], page 510). The skew-list-map employs optimization that’s specific to one-argument case.

12.24 data.sparse - Sparse data containers

data.sparse

This module provides a sparse vector and sparse matrix, a space efficient data container indexed by nonnegative integer(s), and a sparse table, a hash table using a sparse vector as a backing storage.

A sparse vector associates a nonnegative integer index to a value. It has vector in its name since it is indexed by an integer, but it isn’t like a flat array on contiguous memory; it’s more like an associative array. (Internally, the current implementation uses compact trie structure.) It is guaranteed that you can store a value with index at least up to \(2^{32}-1\); the actual maximum bits of indexes can be queried by sparse-vector-max-index-bits. (We have a plan to remove the maximum bits limitation in future).
Unlike ordinary vectors, you don’t need to specify the size of a sparse vector when you create one. You can just set a value to any index in the supported range.

(define v (make-sparse-vector))

(sparse-vector-set! v 0 'a)
(sparse-vector-ref v 0) ⇒ a

(sparse-vector-set! v 100000000 'b)
(sparse-vector-ref v 100000000) ⇒ b

;; set! also work
(set! (sparse-vector-ref v 100) 'c)
(sparse-vector-ref v 100) ⇒ c

If you try to access an element that hasn’t been set, an error is signaled by default. You can set a default value for each vector, or give a fallback value to sparse-vector-ref, to suppress the error.

(sparse-vector-ref v 1) ⇒ error
(sparse-vector-ref v 1 'noval) ⇒ noval

(let1 w (make-sparse-vector #f :default 'x)
(sparse-vector-ref w 1)) ⇒ x

A sparse matrix is like a sparse vector, except it can be indexed by a pair of integers.

A sparse table works just like a hash table, but it uses a sparse vector to store the values using hashed number of the keys.

The main reason of these sparse data containers are for memory efficiency. If you want to store values in a vector but knows you’ll use only some entries sparsely, obviously it is waste to allocate a large vector and to leave many entries unused. But it is worse than that; Gauche’s GC doesn’t like a large contiguous region of memory. Using lots of large vectors adds GC overhead quickly. It becomes especially visible when you store large number of entries (like >100,000) into hash tables, since Gauche’s built-in hash tables use a flat vector as a backing storage. You’ll see the heap size grows quickly and GC runs more frequently and longer. On the other hand, sparse table works pretty stable with large number of entries.

Sparse data containers does have overhead on access speed. They are a bit slower than the ordinary hash tables, and much slower than ordinary vectors. We should note, however, as the number of entries grow, access time on ordinary hash tables grows quicker than sparse tables and eventually two become comparable.

It depends on your application which you should use, and if you’re not sure, you need to benchmark. As a rule of thumb, if you use more than several hashtables each of which contains more than a few tens of thousands of entries, sparse tables may work better. If you see GC Warnings telling “repeated allocation of large blocks”, you should definitely consider sparse tables.

12.24.1 Sparse vectors

<sparse-vector-base> [Class]
{data.sparse} An abstract base class of sparse vectors. Inherits <dictionary> and <collection>. Note that sparse vectors are not <sequence>; even they can be indexable by integers, they don’t have means of ordered access.

Sparse vector may be a general vector that can contain any Scheme objects (like <vector>), or a specialized vector that can contain only certain types of numbers (like <s8vector> etc.).
All of these sparse vectors can be accessed by the same API.

Sparse vectors also implements the Collection API (see Section 9.6 [Collection framework], page 404) and the Dictionary API (see Section 9.10 [Dictionary framework], page 428).

```scheme
<sparse-vector>  [Class]
<sparse-@vector>  [Class]
{data.sparse} The actual sparse vector classes. Inherits <sparse-vector-base>. An instance of <sparse-vector> can contain any Scheme objects.

@ is either one of s8, u8, s16, u16, s32, u32, s64, u64, f16, f32, or f64. The range of values an instance of those classes can hold is the same as the corresponding <@vector> class in gauche.uvector (see Section 6.13.2 [Uniform vectors], page 198). That is, <sparse-u8vector> can have exact integer values between 0 and 255.

make-sparse-vector :optional type :key default
{data.sparse} Creates an empty sparse vector. The type argument can be #f (default), one of subclasses of <sparse-vector-base>, or a symbol of either one of s8, u8, s16, u16, s32, u32, s64, u64, f16, f32, or f64. If type is omitted or #f, a <sparse-vector> is created. If it is a class, an instance of the class is created (It is an error to pass a class that is not a subclass of <sparse-vector-base>.) If it is a symbol, an instance of corresponding <sparse-@vector> is created.

You can specify the default value of the vector by default keyword argument. If given, the vector behaves as if it is filled with the default value (but the vector iterator only picks the values explicitly set).

Note that you have to give the optional argument as well to specify the keyword argument.

(define v (make-sparse-vector 'u8 :default 128))

(sparse-vector-ref v 0) ⇒ 128

sparse-vector-max-index-bits  {data.sparse} Returns maximum number of bits of allowed integer. If this returns 32, the index up to (expt 2 32) is supported. It is guaranteed that this is at least 32.

In the following entries, the argument sv denotes an instance of sparse vector; an error is signaled if other object is passed.

sparse-vector-copy sv  {data.sparse} Returns a copy of a sparse vector sv.

sparse-vector-ref sv k :optional fallback  {data.sparse} Returns k-th element of a sparse vector sv, where k must an exact integer.

If the sparse vector doesn’t have a value for k, it behaves as follows:

- If fallback is given, it is returned.
- Otherwise, if the vector has the default value, it is returned.
- Otherwise, an error is signaled.

sparse-vector-set! sv k value  {data.sparse} Sets value for k-th element of a sparse vector sv. K must be a nonnegative exact integer, and below the maximum allowed index.

If sv is a numeric sparse vector, value must also be within the allowed range, or an error is signaled.

sparse-vector-num-entries sv  {data.sparse} Returns the number of entries in sv.
Chapter 12: Library modules - Utilities

Basic Sparse Vector Functions

- **sparse-vector-exists? sv k**
  - {data.sparse} Returns #t if sv has an entry for index k, #f otherwise.

- **sparse-vector-delete! sv k**
  - {data.sparse} Deletes the k-th entry of sv. If sv had the entry, returns #t. If sv didn’t have the entry, returns #f.

- **sparse-vector-clear! sv**
  - {data.sparse} Empties a sparse vector.

- **sparse-vector-inc! sv k delta :optional (fallback 0)**
  - {data.sparse} This is a shortcut of the following. It is especially efficient for numeric sparse vectors.
    
    \[(sparse-vector-set! sv k (+ (sparse-vector-ref sv k fallback) delta))\]

    If the result of addition exceeds the allowed value range of sv, an error is signaled. In future we’ll allow an option to clamp the result value within the range.

- **sparse-vector-update! sv k proc :optional fallback**
- **sparse-vector-push! sv k val**
- **sparse-vector-pop! sv k :optional fallback**

The following procedures traverses a sparse vector. Note that elements are not visited in the order of index; it’s just like hash table traversers.

At this moment, if you want to walk a sparse vector with increasing/decreasing index order, you have to get a list of keys by sparse-vector-keys, sort it, then use it to retrieve values. We may add an option in future to make-sparse-vector so that those walk operation will be more convenient.

- **sparse-vector-fold sv proc seed**
  - {data.sparse} For each entry in sv, calls proc as \((proc k_n v_n seed_n)\), where \(k_n\) is an index and \(v_n\) is a value for it, and \(seed_n\) is the returned value of the previous call to proc if \(n \geq 1\), and seed if \(n = 0\). Returns the value of the last call of proc.

- **sparse-vector-for-each sv proc**
- **sparse-vector-map sv proc**
  - {data.sparse} Calls proc with index and value, e.g. \((proc k value)\), for each element of sv.

  The results of proc are discarded by sparse-vector-for-each, and gathered to a list and returned by sparse-vector-map.

- **sparse-vector-keys sv**
- **sparse-vector-values sv**
  - {data.sparse} Returns a list of all keys and all values in sv, respectively.

12.24.2 Sparse matrixes

A sparse matrix is like a sparse vector, except it can be indexed by two nonnegative integers.

Note: This implementation of sparse matrixes aims at a reasonable space efficiency for sparse matrixes without knowing its structure beforehand (imagine, for example, a 2D map with some scattered landmarks). If what you want is a sparse matrix implementation for efficient numeric calculations, with certain particular structures, probably the access speed of this module isn’t suitable.
Currently, each index can have half of bits of `sparse-vector-max-index-bits`. We’ll remove this limitation in future.

**<sparse-matrix-base>**

{data.sparse} An abstract base class of sparse matrixes. Inherits `<collection>`. Like sparse vectors, a sparse matrix can be of type that can store any Scheme objects, or that can store only certain types of numbers.

All of these sparse matrix subtypes can be accessed by the same API.

**<sparse-matrix>**

{data.sparse} The actual sparse matrix classes. Inherits `<sparse-matrix-base>`. An instance of `<sparse-matrix>` can contain any Scheme objects.

@ is either one of s8, u8, s16, u16, s32, u32, s64, u64, f16, f32, or f64. The range of values an instance of those classes can hold is the same as the corresponding `<@vector>` class in `gauche.uvector` (see Section 6.13.2 [Uniform vectors], page 198). That is, `<sparse-s@matrix>` can have exact integer values between 0 and 255.

**make-sparse-matrix** 

`:optional type :key default`

{data.sparse} Creates an empty sparse matrix. The type argument can be #f (default), one of subclasses of `<sparse-matrix-base>`, or a symbol of either one of s8, u8, s16, u16, s32, u32, s64, u64, f16, f32, or f64.

If type is omitted or #f, a `<sparse-matrix>` is created. If it is a class, an instance of the class is created (It is an error to pass a class that is not a subclass of `<sparse-matrix-base>`.) If it is a symbol, an instance of corresponding `<sparse-s@matrix>` is created.

You can specify the default value of the matrix by default keyword argument. If given, the vector behaves as if it is filled with the default value (but the matrix iterator only picks the values explicitly set).

Note that you have to give the optional argument as well to specify the keyword argument.

**sparse-matrix-num-entries** mat

{data.sparse} Returns the number of entries explicitly set in a sparse matrix mat.

**sparse-matrix-ref** mat x y :optional fallback

{data.sparse} Returns an element indexed by (x, y) in a sparse matrix mat. If the indexed element isn’t set, fallback is returned if provided; otherwise, if the matrix has the default value, it is returned; otherwise, an error is raised.

**sparse-matrix-set!** mat x y value

{data.sparse} Set value to the sparse matrix mat at the location (x, y).

**sparse-matrix-exists?** mat x y

{data.sparse} Returns #t iff the sparse matrix mat has a value at (x, y).

**sparse-matrix-clear!** mat

{data.sparse} Empties the sparse matrix mat.

**sparse-matrix-delete!** mat x y

{data.sparse} Remove the value at (x, y) from the sparse matrix mat.

**sparse-matrix-copy** mat

{data.sparse} Returns a fresh copy of mat.
Chapter 12: Library modules - Utilities

sparse-matrix-update! mat x y proc :optional fallback
{data.sparse} Call proc with the value at (x, y) of the sparse matrix, and sets the result of proc as the new value of the location.

The optional fallback argument works just like sparse-matrix-ref; if provided, it is passed to proc in case the matrix doesn’t have a value at (x, y). If fallback isn’t provided and the matrix doesn’t have a value at the location, the default value of the matrix is used if it has one. Otherwise, an error is signalled.

sparse-matrix-inc! mat x y delta :optional fallback
{data.sparse}
(sparse-matrix-update! mat x y (cut + <> delta) fallback)

sparse-matrix-push! mat x y val
{data.sparse}
(sparse-matrix-update! mat x y (cut cons val <>) '())

sparse-matrix-pop! mat x y
{data.sparse}
(rlet1 r #f
(sparse-matrix-update! mat x y (^p (set! r (car p)) (cdr p)))))

sparse-matrix-fold mat proc seed
{data.sparse} Loop over values in the sparse matrix mat. The procedure proc is called with four arguments, x, y, val and seed, for each index (x, y) which has the value val. The initial value of seed is the one given to sparse-matrix-fold, and the result of proc is passed as the next seed value. The last result of proc is returned from sparse-matrix-fold.

The procedure proc is only called on the entries that actually has a value, and the order of which the procedure is called is undefined.

sparse-matrix-map mat proc
{data.sparse}
(sparse-matrix-fold sv (^[x y v s] (cons (proc x y v) s)) '())

sparse-matrix-for-each mat proc
{data.sparse}
(sparse-matrix-fold sv (^[x y v _] (proc x y v)) #f)

sparse-matrix-keys mat
{data.sparse}
(sparse-matrix-fold sv (^[x y _ s] (cons (list x y) s)) '())

sparse-matrix-values mat
{data.sparse}
(sparse-matrix-fold sv (^[x y v s] (cons v s)) '())

12.24.3 Sparse tables

<sparse-table>
{data.sparse} A class for sparse table. Inherits <dictionary> and <collection>.

Operationally sparse tables are the same as hash tables, but the former consumes less memory in trade of slight slower access. (Roughly x1.5 to x2 access time when the table is small. As the table gets larger the difference becomes smaller.)
**make-sparse-table**  
*comparator*

Creates and returns an empty sparse table. The *comparator* argument specifies how to compare and hash keys; it must be either a comparator (see Section 6.2.4 [Basic comparators], page 116), or one of the symbols eq?, eqv?, equal? and string=?, like hash tables (see Section 6.14.1 [Hashables], page 206). If it is a symbol, eq-comparator, eqv-comparator, equal-comparator or string-comparator are used, respectively.

**sparse-table-comparator**  
*st*

Returns the comparator used in the sparse table *st*.

**sparse-table-copy**  
*st*

Returns a copy of a sparse table *st*.

**sparse-table-num-entries**  
*st*

Returns the number of entries in a sparse table *st*.

**sparse-table-ref**  
*st key :optional fallback*

Retrieves a value associated to the *key* in *st*. If no entry with *key* exists, *fallback* is returned when it is provided, or an error is signaled otherwise.

**sparse-table-set!**  
*st key value*

Sets *value* with *key* in *st*.

**sparse-table-exists?**  
*st key*

Returns #t if an entry with *key* exists in *st*, #f otherwise.

**sparse-table-delete!**  
*st key*

Deletes an entry with *key* in *st* if it exists. Returns #t if an entry is actually deleted, or #f if there hasn’t been an entry with *key*.

**sparse-table-clear!**  
*st*

Empties *st*.

**sparse-table-update!**  
*st key proc :optional fallback*

**sparse-table-push!**  
*st key val*

**sparse-table-pop!**  
*st key :optional fallback*

**sparse-table-fold**  
*st proc seed*

**sparse-table-for-each**  
*st proc*

**sparse-table-map**  
*st proc*

**sparse-table-keys**  
*st*

**sparse-table-values**  
*st*

### 12.25 data.trie - Trie

This module provides *Trie*, a dictionary-like data structure that maps keys to values, where a key is an arbitrary sequence. Internally it stores the data as a tree where each node corresponds to each element in the key sequence. Key lookup is O(n) where n is the length of the key, and not affected much by the number of total entries. Also it is easy to find a set of values whose keys share a common prefix.
Chapter 12: Library modules - Utilities

The following example may give you the idea.

```
(define t (make-trie)) ;; create a trie
(trie-put! t "pho" 3) ;; populate the trie
(trie-put! t "phone" 5)
(trie-put! t "phrase" 6)

(trie-get t "phone") ⇒ 5 ;; lookup
(trie-common-prefix t "pho") ;; common prefix search
⇒ ("phone" . 5) ("pho" . 3)
(trie-common-prefix-keys t "ph")
⇒ ("phone" "pho" "phrase")
```

Tries are frequently used with string keys, but you are not limited to do so; any sequence (see Section 9.31 [Sequence framework], page 510) can be a key. If the types of keys differ, they are treated as different keys:

```
(trie-put! t '(#\p #\h #\o) 8) ;; different key from "pho"
```

Trie inherits <collection> and implements collection framework including the builder. So you can apply generic collection operations on a trie (see Section 9.6 [Collection framework], page 404). When iterated, each element of a trie appears as a pair of a key and a value.

```
<trie> [Class]
{data.trie} A class for Trie. No slots are intended for public. Use the following procedures to operate on tries.

This class also implements the dictionary interface (see Section 9.10.1 [Generic functions for dictionaries], page 428).

make-trie :optional tab-make tab-get tab-put! tab-fold tab-empty? [Function]
{data.trie} Creates and returns an empty trie. The optional arguments are procedures to customize how the nodes of the internal tree are managed.

Each node can have a table to store its child nodes, indexed by an element of the key sequence (e.g. if the trie uses strings as keys, a node’s table is indexed by characters).

1. **tab-make** A procedure with no arguments. When called, creates and returns an empty table for a node.

2. **tab-get** tab elt

   Returns a child node indexed by elt, or returns #f if the table doesn’t have a child for elt.

3. **tab-put!** tab elt child-node

   If child-node isn’t #f, stores a child-node with index elt. If child-node is #f, removes the entry with index elt. In both cases, this procedure should return the updated table.

4. **tab-fold** tab proc seed

   Calls proc for every index and node in tab, while passing a seed value, whose initial value is seed. That is, proc has a type of (index, node, seed) -> seed. Should return the last result of proc.

5. **tab-empty?** tab

   Returns #t if tab is empty, #f otherwise. You can omit or pass #f to this procedure; then we use tab-fold to check if tab is empty, which can be expensive.
The default assumes `eqv?`-hashtables, i.e. the following procedures are used.

```scheme
(tab-make: (lambda () (make-hash-table 'eqv?)))
(tab-get: (lambda (tab k) (hash-table-get tab k #f)))
(tab-put!: (lambda (tab k v)
    (if v
        (hash-table-put! tab k v)
        (hash-table-delete! tab k))
    tab))
(tab-fold: hash-table-fold)
(tab-empty?: (lambda (tab) (zero? (hash-table-num-entries tab)))
```

The following example creates a trie using assoc list to manage children, while comparing string keys with case-insensitive way:

```scheme
(make-trie list
    (cut assoc-ref <> <> #f char-ci=?)
    (lambda (t k v)
        (if v
            (assoc-set! t k v char-ci=?)
            (alist-delete! k t char-ci=?)))
    (lambda (t f s) (fold f s t))
    null?)
```

It is important that `tab-put!` must return an updated table—by that, you can replace the table structure on the fly. For example, you may design a table which uses assoc list when the number of children are small, and then switches to a vector (indexed by character code) once the number of children grows over a certain threshold.

```scheme
trie params kv ...
```

A convenient version of `trie` when you only concern the keys. Each value is the same as its key. The following example creates a trie with two entries and the default table procedures.

```scheme
(trie '() '("foo" . a) '("bar" . b))
```

```scheme
trie-with-keys params key ...
```

A convenient version of `trie` when you only concern the keys. Each value is the same as its key. The following example creates a trie with two entries and the default table procedures.

```scheme
(trie-with-keys '() "foo" "bar")
```

```scheme
trie? obj
```

Returns `#t` if `obj` is a trie, or `#f` otherwise.

```scheme
trie-num-entries trie
```

Returns the number of entries in `trie`.

```scheme
trie-exists? trie key
```

Returns `#t` if `trie` contains an entry with `key`, or returns `#f` otherwise.

```scheme
(let1 t (trie '() '("foo" . ok))
    (list (trie-exists? t "foo")))
```
(trie-exists? t "fo")
(trie-exists? t "bar"))
⇒ '(#t #f #f)

trie-partial-key? trie seq
{data.trie} Returns #t if there’s at least one key in trie that is not equal to seq but seq matches its prefix. Note that seq may or may not a key of trie; see the example below.

(define t (trie '() '("foo" . ok) '("fo" . ok)))

(trie-partial-key? t "f") ⇒ #t
(trie-partial-key? t "fo") ⇒ #t
(trie-partial-key? t "foo") ⇒ #f
(trie-partial-key? t "bar") ⇒ #f

trie-get trie key :optional fallback
{data.trie} Returns the value associated with key in trie, if such an entry exists. When there’s no entry for key, if fallback is given, it is returned; otherwise, an error is signaled.

trie-put! trie key value
{data.trie} Puts value associated to key into trie.

trie-update! trie key proc :optional fallback
{data.trie} Works like the following code, except that the lookup of entry in trie is done only once.

(let ((val (trie-get trie key fallback)))
  (trie-put! trie key (proc val)))

trie-delete! trie key
{data.trie} Removes an entry associated with key from trie. If there’s no such entry, this procedure does nothing.

trie->list trie
{data.trie} Makes each entry in trie to a pair (key . value) and returns a list of pairs of all entries. The order of entries are undefined.

trie-keys trie
{data.trie} Returns a list of all keys in trie, respectively. The order of keys/values are undefined.

trie-values trie
{data.trie} Returns a list of all values in trie, respectively. The order of keys/values are undefined.

trie->hash-table trie ht-type
{data.trie} Creates a hash table with type ht-type (see Section 6.14.1 [Hashtables], page 206, about hash table types), and populates it with every key and value pair in trie.

trie-longest-match trie seq :optional fallback
{data.trie} Returns a pair of the key and its value, where the key is the longest prefix of seq. If no such key is found, fallback is returned if it is provided, or an error is thrown. Do not confuse this with trie-common-prefix-* procedures below; In this procedure, the key is the prefix of the given argument. In trie-common-prefix-* procedures, the given argument is the prefix of the keys.

(let1 t (make-trie)
  (trie-put! t "a" 'a)
  (trie-put! t "ab" 'ab)
(trie-longest-match t "abc") ⇒ ("ab" . ab)
(trie-longest-match t "acd") ⇒ ("a" . a)
(trie-longest-match t "ab") ⇒ ("ab" . ab)
(trie-longest-match t "zy") ⇒ error

trie-common-prefix trie prefix
trie-common-prefix-keys trie prefix
trie-common-prefix-values trie prefix

{data.trie} Gather all entries whose keys begin with prefix; trie-common-prefix returns those entries in a list of pairs (key . value); trie-common-prefix-keys returns a list of keys; and trie-common-prefix-values returns a list of values. The order of entries in a returned list is undefined. If trie contains no entry whose key has prefix, an empty list is returned.

Note that prefix matching doesn’t consider the type of sequence; if trie has entries for "foo" and (# #\o #\o), (trie-common-prefix trie "foo") will return both entries.

trie-common-prefix-fold trie prefix proc seed

{data.trie} For each entry whose key begins with prefix, calls proc with three arguments, the entry’s key, its value, and the current seed value. Seed is used for the first seed value, and the value proc returns is used for the seed value of the next call of proc. The last returned value from proc is returned from trie-common-prefix-fold. The order of entries on which proc is called is undefined. If trie contains no entry whose key has prefix, proc is never called and seed is returned.

trie-common-prefix-map trie prefix proc

{data.trie} These are to trie-common-prefix-fold as map and for-each are to fold; trie-common-prefix-map calls proc with key and value for matching entries and gathers its result to a list; trie-common-prefix-for-each also applies proc, but discards its results.

trie-fold trie proc seed

trie-map trie proc

trie-for-each trie proc

{data.trie} These procedures are like their common-prefix versions, but traverse entire trie instead.

12.26 data.ulid - Universally unique lexicographically sortable identifier

data.ulid

This module implements ULID (https://github.com/ulid/spec), a UUID-like unique identifier that can be sortable in chronological order and has slightly more compact string representation.

(define gen-ulid (make-ulid-generator))

(gen-ulid) ⇒ #<ulid 01G70P3EWJT28VPQ12NTFFDSZ1>

(ulid->string (gen-ulid)) ⇒ "01G70P3Y72TQ1HYY90G8TJJSY07"

make-ulid-generator :optional random-source

Creates and returns a generator that yields a fresh ULID object every time it is called. The optional random-source argument is a SRFI-27 random source (see Section 11.8 [Sources of
random bits], page 706) to be used for the random part of ULID. When omitted, an internal
random source is used.

To guarantee uniqueness and monotonicity, a ULID generator needs to keep internal state.
If you want to share the generator among multiple threads, you need to manage your own
lock. Keeping the generator private in one thread eliminates the need of mutex (however,
monotonicity is only guaranteed within each generator).

A ULID consists of 48bit timestamp field (milliseconds since Unix epoch) and 80bit ran-
domness field. The randomness field are generated randomly if it is the first ULID within
the timestamp. If more than one ULID needs to be generated in a millisecond window, the
subsequent ULID gets randomness field incremented from the previous ULID, so that the
chronological order is maintained.

The ULID spec says if the randomness field overflows because so many ULIDs are generated
within a millisecond, it is an error. Such scenario is very unlikely but can occur. In our
implementation, however, we wait for the next millisecond boundary in case we ran out
randomness field, so the generator never yields an error.

ulid? obj
Returns #t iff obj is an ULID object.

ulid-timestamp ulid
Returns the timestamp part of ulid (48bits) as an exact integer.

ulid-randomness ulid
Returns the randomness part of ulid (80bits) as an exact integer.

ulid=? ulid1 ulid2
Returns true if two ULIDs are equivalent.

ulid<? ulid1 ulid2
Returns true if ulid1 precedes ulid2. If both ULIDs are created from the same generator,
this means ulid1 is created chronologically before ulid2. If they are created from different
generators, the chronological order is not guaranteed, but the discrepancy only occurs when
two ULIDs are created at the same timestamp in millisecond resolution.

ulid-hash ulid
Returns a hash value of ulid in a nonnegative exact integer.

ulid-comparator
Bound to a comparator suitable to compare and hash ULIDs.

ulid->bytevector ulid
Convert between ulid and a u8vector of length 16. Bits are packed in big-endian order (first
8 bits of timestamp to first byte, and so on).

ulid->integer ulid
Convert between ulid and an exact nonnegative integer up to 128bit.

ulid->string ulid
Convert ulid to a string of Base32 representation; the digits are
0123456789ABCDEFHJKLMNPQRSTVWXYZ (I, L, O, and U are excluded to avoid
confusion). Length of the string is always 26.

ulid->string always use uppercase characters; on the other hand, string->ulid allows
both lowercase and uppercase characters.
12.27 dbi - Database independent access layer

dbi [Module]

This module provides the unified interface to access various relational database systems (RDBMS). The operations specific to individual database systems are packaged in database driver (DBD) modules, which is usually loaded implicitly by DBI layer.

The module is strongly influenced by Perl's DBI/DBD architecture. If you have used Perl DBI, it would be easy to use this module.

It's better to look at the example. This is a simple outline of accessing a database by dbi module:

```perl
(use dbi)
(use gauche.collection) ; to make 'map' work on the query result

(guard (e ((<dbi-error> e)
               ;; handle error
               )))
(let* ((conn (dbi-connect "dbi:mysql:test;host=dbhost"))
       (query (dbi-prepare conn
               "SELECT id, name FROM users WHERE department = ?"))
       (result (dbi-execute query "R&D"))
       (getter (relation-accessor result)))
(map (lambda (row)
       (list (getter row "id")
             (getter row "name"))))
```

There's nothing specific to the underlying database system except the argument "dbi:mysql:test;host=dbhost" passed to dbi-connect, from which dbi module figures out that it is an access to mysql database, loads dbd.mysql module, and let it handle the mysql-specific stuff. If you want to use whatever database system, you can just pass "dbi:whatever:parameter" to dbi-connect instead, and everything stays the same as far as you have dbd.whatever installed in your system.

A query to the database can be created by dbi-prepare. You can issue the query by dbi-execute. This two-phase approach allows you to create a prepared query, which is a kind of parameterized SQL statement. In the above example the query takes one parameter, denoted as '?' in the SQL. The actual value is given in dbi-execute. When you issue similar queries a lot, creating a prepared query and execute it with different parameters may give you performance gain. Also the parameter is automatically quoted.

When the query is a SELECT statement, its result is returned as a collection that implements the relation protocol. See Section 9.6 [Collection framework], page 404, and Section 12.89 [Relation framework], page 1031, for the details.

The outermost guard is to catch errors. The dbi related errors are supposed to inherit <dbi-error> condition. There are a few specific errors defined in dbi module. A specific dbd layer may define more specific errors.

In the next section we describe user-level API, that is, the procedures you need to concern when you're using dbi. The following section is for the driver API, which you need to use to write a specific dbd driver to make it work with dbi framework.

12.27.1 DBI user API
DBI Conditions

There are several predefined conditions dbi API may throw. See Section 6.19 [Exceptions], page 245, for the details of conditions.

<dbi-error> [Condition Type]
{dbi} The base class of dbi-related conditions. Inherits <error>.

<dbi-nonexistent-driver-error> [Condition Type]
{dbi} This condition is thrown by dbi-connect when it cannot find the specified driver. Inherits <dbi-error>.

driver-name [Instance Variable of <dbi-nonexistent-driver-error>]
Holds the requested driver name as a string.

<dbi-unsupported-error> [Condition Type]
{dbi} This condition is thrown when the called method isn’t supported by the underlying driver. Inherits <dbi-error>.

<dbi-parameter-error> [Condition Type]
{dbi} This condition is thrown when the number of parameters given to the prepared query doesn’t match the ones in the prepared statement.

Besides these errors, if a driver relies on dbi to parse the prepared SQL statement, <sql-parse-error> may be thrown if an invalid SQL statement is passed to dbi-prepare. (see Section 12.77 [SQL parsing and construction], page 1011).

Connecting to the database

dbi-connect dsn :key username password [Function]
{dbi} Connect to a database using a data source specified by dsn (data source name). Dsn is a string with the following syntax:

    dbi:driver:options

Driver part names a specific driver. You need to have the corresponding driver module, dbd.driver, installed in your system. For example, if dsn begins with "dbi:mysql:“, dbi-connect tries to load dbd.mysql.

Interpretation of the options part is up to the driver. Usually it is in the form of key1=value1;key2=value2;... , but some driver may interpret it differently. For example, mysql driver allows you to specify a database name at the beginning of options. You have to check out the document of each driver for the exact specification of options.

The keyword arguments gives extra information required for connection. The username and password are commonly supported arguments. The driver may recognize more keyword arguments.

If a connection to the database is successfully established, a connection object (an instance of a subclass of <dbi-connection>) is returned. Otherwise, an error is signaled.

<dbi-connection> [Class]
{dbi} The base class of a connection to a database system. Each driver defines a subclass of this to keep information about database-specific connections.

dbi-open? (c <dbi-connection>) [Method]
{dbi} Queries whether a connection to the database is still open (active).
**dbi-close** *(c <dbi-connection>)*

[Method]
{dbi} Closes a connection to the database. This causes releasing resources related to this connection. Once closed, c cannot be used for any dbi operations (except passing to **dbi-open**). Calling **dbi-close** on an already closed connection has no effect.

Although a driver usually closes a connection when <dbi-connection> object is garbage-collected, it is not a good idea to rely on that, since the timing of GC is unpredictable. The user program must make sure that it calls **dbi-close** at a proper moment.

**dbi-list-drivers**

[Function]
{dbi} Returns a list of module names of known drivers.

**<dbi-driver>**

[Class]
{dbi} The base class of a driver. You usually don’t need to see this as far as you’re using the high-level **dbi** API.

**dbi-make-driver** *driver-name*

[Function]
{dbi} This is a low-level function called from **dbi-connect** method, and usually a user doesn’t need to call it.

Loads a driver module specified by *driver-name*, and instantiate the driver class and returns it.

### Preparing and issuing queries

**dbi-prepare** *conn sql :key pass-through ...

{dbi} From a string representation of SQL statement *sql*, creates and returns a query object (an instance of <dbi-query> or its subclass) for the database connection *conn*.

*Sql* may contain parameter slots, denoted by ?.

```
(dbi-prepare conn "insert into tab (col1, col2) values (?, ?)"

(dbi-prepare conn "select * from tab where col1 = ?")
```

They will be filled when you actually issue the query by **dbi-execute**. There are some advantages of using parameter slots: (1) The necessary quoting is done automatically. You don’t need to concern about security holes caused by improper quoting, for example. (2) Some drivers support a feature to send the template SQL statement to the server at the preparation stage, and send only the parameter values at the execution stage. It would be more efficient if you issue similar queries lots of time.

If the backend doesn’t support prepared statements (SQL templates having ? parameters), the driver may use **text.sql** module to parse *sql*. It may raise **sql-parse-error** condition if the given SQL is not well formed.

You may pass a true value to the keyword argument **pass-through** to suppress interpretation of SQL and pass *sql* as-is to the back end database system. It is useful if the back-end supports extension of SQL which **text.sql** doesn’t understand.

If the driver lets prepared statement handled in back-end, without using **text.sql**, the **pass-through** argument may be ignored. The driver may also take other keyword arguments. Check out the documentation of individual drivers.

*Note:* Case folding of SQL statement is implementation dependent. Some DBMS may treat table names and column names in case insensitive way, while others do in case sensitive way. To write a portable SQL statement, make them quoted identifiers, that is, always surround names by double quotes.

**<dbi-query>**

{dbi} Holds information about prepared query, created by **dbi-prepare**. The following slots are defined.
connection
Contains the <dbi-connection> object.

prepared
If the driver prepares query by itself, this slot may contain a prepared statement. It is up
to each driver how to use this slot, so the client shouldn’t rely on its value.

\[\text{Method} \quad \text{dbi-open?} (q <\text{dbi-query}>) \] 
{dbi} Returns #t iff the query can still be passed to dbi-execute.

\[\text{Method} \quad \text{dbi-close} (q <\text{dbi-query}>) \] 
{dbi} Destroy the query and free resources associated to the query. After this operation,
dbi-open? returns #f for q, and the query can’t be used in any other way. Although the
resource may be freed when q is garbage-collected, it is strongly recommended that the
application closes queries explicitly.

\[\text{Method} \quad \text{dbi-execute} (q <\text{dbi-query}>) \text{ parameter } \ldots \] 
{dbi} Executes a query created by dbi-prepare. You should pass the same number of
parameters as the query expects.

If the issued query is select statement, dbi-execute returns an object represents a relation.
A relation encapsulates the values in rows and columns, as well as meta information like
column names. See "Retrieving query results" below for how to access the result.

If the query is other types, such as create, insert or delete, the return value of the query
closure is unspecified.

\[\text{Method} \quad \text{dbi-do} \text{ conn sql :optional options parameter-value } \ldots \] 
{dbi} This is a convenience procedure when you create a query and immediately execute it.
It is equivalent to the following expression, although the driver may overload this method to
avoid creating intermediate query object to avoid the overhead.

\[(\text{dbi-execute (apply dbi-prepare conn sql options)} \text{ parameter-value } \ldots)\]

\[\text{Method} \quad \text{dbi-escape-sql} \text{ conn str} \] 
{dbi} Returns a string where special characters in str are escaped.
The official SQL standard only specify a single quote (’) as such character. However, it
doesn’t specify non-printable characters, and the database system may use other escaping
characters. So it is necessary to use this method rather than doing escaping by your own.

\[
;; \text{ assumes c is a valid DBI connection} \\
(dbi-escape-sql c "don’t know") \\
⇒ "don’t know"
\]

Retrieving query results
If the query is a select statement, it returns an object of both <collection> and <relation>. It
is a collection of rows (that is, it implements <collection> API), so you can use map,
for-each or other generic functions to access rows. You can also use the relation API to retrieve
column names and accessors from it. See Section 12.89 [Relation framework], page 1031, for the
relation API, and Section 9.6 [Collection framework], page 404, for the collection API.

The actual class of the object returned from a query depends on the driver, but you may use
the following method on it.

\[\text{Method} \quad \text{dbi-open? result} \] 
{dbi} Check whether the result of a query is still active. The result may become inactive
when it is explicitly closed by dbi-close and/or the connection to the database is closed.
dbi-close result

{dbi} Close the result of the query. This may cause releasing resources related to the result. You can no longer use result once it is closed, except passing it to dbi-open?

Although a driver usually releases resources when the result is garbage-collected, the application shouldn’t rely on that and is recommended call dbi-close explicitly when it is done with the result.

12.27.2 Writing drivers for DBI

Writing a driver for a specific database system means implementing a module dbd.foo, where foo is the name of the driver.

The module have to implement several classes and methods, as explained below.

**DBI classes to implement**

You have to define the following classes.

- Subclass <dbi-driver>. The class name must be <foo-driver>, where foo is the name of the driver. Usually this class produces a singleton instance, and is only used to dispatch dbi-make-connection method below.
- Subclass <dbi-connection>. An instance of this class is created by dbi-make-connection. It needs to keep the information about the actual connections.
- Subclass <relation> and <collection> to represent query results suitable for the driver. (In most cases, the order of the result of SELECT statement is significant, since it may be sorted by ORDER BY clause. Thus it is more appropriate to inherit <sequence>, rather than <collection>).
- Optionally, subclass <dbi-query> to keep driver-specific information of prepared queries.

**DBI methods to implement**

The driver need to implement the following methods.

dbi-make-connection (d <foo-driver>) (options <string>) (options-alist <list>) :key username password . . .

{dbi} This method is called from dbi-connect, and responsible to connect to the database and to create a connection object. It must return a connection object, or raise an <dbi-error> if it cannot establish a connection.

Options is the option part of the data source name (DSN) given to dbi-connect. options-alist is an assoc list of the result of parsing options. Both are provided so that the driver may interpret options string in nontrivial way.

For example, given "dbi:foo:myaddressbook;host=dbhost;port=8998" as DSN, foo’s dbi-make-connection will receive "myaddressbook;host=dbhost;port=8998" as options, and (("myaddressbook" . #t) ("host" . "dbhost") ("port" . "8998")) as options-alist.

After options-alist, whatever keyword arguments given to dbi-connect are passed. DBI protocol currently specifies only username and password. The driver may define other keyword arguments. It is recommended to name the driver-specific keyword arguments prefixed by the driver name, e.g. for dbd.foo, it may take a :foo-whatever keyword argument.

It is up to the driver writer to define what options are available and the syntax of the options. The basic idea is that the DSN identifies the source of the data; it’s role is like URL in WWW. So, it may include the hostname and port number of the database, and/or the name of the database, etc. However, it shouldn’t include information related to authentication, such as username and password. That’s why those are passed via keyword arguments.
Method\ndbi-prepare\n\n(c <foo-connection>) (sql <string>) :key pass-through . . .

{dbi} This method should create and return a prepared query object, which is an instance of \n<dbi-query> or its subclass. The query specified by sql is issued to the database system when the prepared query object is passed to \ndbi-execute.

The method must set c to the connection slot of the returned query object.

Sql is an SQL statement. It may contain placeholders represented by '?' . The query closure should take the same number of arguments as of the placeholders. It is up to the driver whether it parses sql internally and construct a complete SQL statement when the query closure is called, or it passes sql to the back-end server to prepare the statement and let the query closure just send parameters.

If the driver parses SQL statement internally, it should recognize a keyword argument\npass-through. If a true value is given, the driver must treat sql opaque and pass it as is when the query closure is called.

The driver may define other keyword arguments. It is recommended to name the driver-specific keyword arguments prefixed by the driver name, e.g. for dbd.foo, it may take a\n:foo-whatever keyword argument.

Method\ndbi-execute-using-connection\n\n(c <foo-connection>) (q <dbi-query>) (params <list>)

{dbi} This method is called from \ndbi-execute. It must issue the query kept in q. If the query is parameterized, the actual parameters given to \ndbi-execute are passed to params argument.

If q is a select-type query, this method must return an appropriate relation object.

Method\ndbi-escape-sql\n\n(c <foo-connection>) str

{dbi} If the default escape method isn’t enough, the driver may overload this method to implement a specific escaping. For example, MySQL treats backslash characters specially as well as single quotes, so it has its \ndbi-escape-sql method.

Method\ndbi-open?\n\n(c <foo-connection>) (q <foo-query>) (r <foo-result>)

{dbi} Queries open/close status of a connection and a result, and closes a connection and a result. The close methods should cause releasing resources used by connection/result. The driver has to allow \ndbi-close to be called on a connection or a result which has already been closed.

Method\ndbi-do\n\n(c <foo-connection>) (sql <string>) :optional options parameter-value . . .

{dbi} The default method uses \ndbi-prepare and \ndbi-execute to implement the function. It just works, but the driver may overload this method in order to skip creating intermediate query object for efficiency.

DBI utility functions

The following functions are low-level utilities which you may use to implement the above methods.

Function\ndbi-parse-dsn data-source-name

{dbi} Parse the data source name (DSN) string given to \ndbi-connect, and returns tree values: (1) The driver name in a string. (2) 'options' part of DSN as a string. (3) parsed
options in an assoc list. This may raise `<dbi-error>` if the given string doesn’t conform DSN syntax.

You don’t need to use this to write a typical driver, for the parsing is done before `dbi-make-connection` is called. This method may be useful if you’re writing a kind of meta-driver, such as a proxy.

`dbi-prepare-sql connection sql` [Function]

 `{dbi}` Parses an SQL statement `sql` which may contain placeholders, and returns a closure, which generates a complete SQL statement when called with actual values for the parameters. If the back-end doesn’t support prepared statements, you may use this function to prepare queries in the driver.

`Connection` is a DBI connection to the database. It is required to escape values within SQL properly (see `dbi-escape-sql` above).

```scheme
;; assume c contains a valid dbi connection
((dbi-prepare-sql c "select * from table where id=?") "foo\'bar")
=> "select * from table where id=\"foo\’\'bar\""
```

12.28 dbm - Generic DBM interface

dbm [Module]

DBM-like libraries provides an easy way to store values to a file, indexed by keys. You can think it as a persistent associative memory.

This modules defines `<dbm>` abstract class, which has a common interface to use various DBM-type database packages. As far as you operate on the already opened database, importing `dbm` module is enough.

To create or open a database, you need a concrete implementation of the database. With the default build-time configuration, the following implementations are included in Gauche. Bindings to various other dbm-like libraries are available as extension packages. Each module defines its own low-level accessing functions as well as the common interface. Note that your system may not have one or more of those DBM libraries; Gauche defines only what the system provides.

`dbm.fsdbm` file-system dbm (see Section 12.29 [File-system dbm], page 876).

`dbm.gdbm` GDBM library (see Section 12.30 [GDBM interface], page 876).

`dbm.ndbm` NDBM library (see Section 12.31 [NDBM interface], page 878).

`dbm.odbm` DBM library (see Section 12.32 [Original DBM interface], page 879).

The following code shows a typical usage of the database.

```scheme
(use dbm) ; dbm abstract interface
(use dbm.gdbm) ; dbm concrete interface

; open the database
(define *db* (dbm-open <gdbm> :path "mydb" :rw-mode :write))

; put the value to the database
(dbm-put! *db* "key1" "value1")

; get the value from the database
(define val (dbm-get *db* "key1"))
```
iterate over the database
(dbm-for-each *db* (lambda (key val) (foo key val)))

; close the database
(dbm-close *db*)

The <dbm> abstract class implements collection and dictionary framework. (See Section 9.6 [Collection framework], page 404, and Section 9.10 [Dictionary framework], page 428, respectively).

12.28.1 Opening and closing a dbm database

$dbm$  
{dbm} An abstract class for dbm-style database. Inherits <dictionary> (see Section 9.10 [Dictionary framework], page 428). Defines the common database operations. This class has the following instance slots. They must be set before the database is actually opened by dbm-open.

The concrete class may add more slots for finer control on the database, such as locking.

path  
Pathname of the dbm database. Some dbm implementation may append suffixes to this.

rw-mode  
Specifies read/write mode. Can be either one of the following keywords:

:read The database will be opened in read-only mode. The database file must exist when dbm-open is called.

:write The database will be opened in Read-write mode. If the database file does not exist, dbm-open creates one.

:create The database will be created and opened in Read-write mode. If the database file exists, dbm-open truncates it.

file-mode  
Specifies the file permissions (as sys-chmod) to create the database. The default value is #o664.

key-convert  
value-convert  
By default, you can use only strings for both key and values. With this option, however, you can specify how to convert other Scheme values to/from string to be stored in the database. The possible values are the followings:

#f The default value. Keys (values) are not converted. They must be a string.

#t Keys (values) are converted to its string representation, using write, to store in the database, and converted back to Scheme values, using read, to retrieve from the database. The data must have an external representation that can be read back. (But it is not checked when the data is written; you’ll get an error when you read the data). The key comparison is done in the string level, so the external representation of the same key must match.

a list of two procedures
Both procedure must take a single argument. The first procedure must receive a Scheme object and returns a string. It is used to convert the keys (values) to
store in the database. The second procedure must receive a string and returns
a Scheme object. It is used to convert the stored data in the database to a
Scheme object. The key comparison is done in the string level, so the external
representation of the same key must match.

<dbm-meta>
{dbm} A metaclass of <dbm> and its subclasses.

`dbm-open (dbm <dbm>)`
{dbm} Opens a dbm database. `dbm` must be an instance of one of the concrete classes that
derived from the <dbm> class, and its slots must be set appropriately. On success, it returns
the `dbm` itself. On failure, it signals an error.

`dbm-open (dbm-class <dbm-meta>) options . . .`
{dbm} A convenient method that creates dbm instance and opens it. It is defined as follows.

(define-method dbm-open ((class <class>) . initargs)
  (dbm-open (apply make class initargs)))

Database file is closed when it is garbage collected. However, to ensure the modification is
properly synchronized, you should close the database explicitly.

`dbm-close (dbm <dbm>)`
{dbm} Closes a database `dbm`. Once the database is closed, any operation to access the
database content raises an error.

`dbm-closed? (dbm <dbm>)`
{dbm} Returns true if a database `dbm` is already closed, false otherwise.

`dbm-type->class dbmtype`
{dbm} Sometimes you don’t know which type of dbm implementation you need to use in
your application beforehand, but rather you need to determine the type according to the
information given at run-time. This procedure fulfills the need.

The `dbmtype` argument is a symbol that names the type of dbm implementation; for example,
gdbm for `dbm.gdbm`, and fsdbm for `dbm.fsdbm`. We assume that the dbm implementation of
type `foo` is provided as a module `dbm.foo`, and its class is named as `<foo>`.

This procedure first checks if the required module has been loaded, and if not, it tries to
load it. If the module loads successfully, it returns the class object of the named dbm
implementation. If it can’t load the module, or can’t find the dbm class, this procedure
returns #f.

(use dbm)

(dbm-type->class 'gdbm)
⇒ #<class <gdbm>>

(dbm-type->class 'nosuchdbm)
⇒ #f

12.28.2 Accessing a dbm database

Once a database is opened, you can use the following methods to access individual key/value
pairs.

`dbm-put! (dbm <dbm>) key value`
{dbm} Put a value with key.
12.28.3 Iterating on a dbm database

To walk over the entire database, following methods are provided.

\[
\text{dbm-fold} \ (dbm <dbm>) \ \text{procedure knil}
\]

{dbm} The basic iterator. For each key/value pair, \text{procedure} is called as \((\text{procedure} \ \text{key} \ \text{value} \ \text{r})\), where \text{r} is \text{knil} for the fist call of \text{procedure}, and the return value of the previous call for subsequent calls. Returns the result of the last call of \text{procedure}. If no data is in the database, \text{knil} is returned.

The following method returns the sum of all the integer values.

\[
\text{dbm-fold} \ (\lambda (k \ v \ r) \ (if \ (\text{integer?} \ v) \ (+ \ v \ r) \ r)) \ 0
\]

\[
\text{dbm-for-each} \ (dbm <dbm>) \ \text{procedure}
\]

{dbm} For each key/value pair in the database \text{dbm}, \text{procedure} is called. Two arguments are passed to \text{procedure}—a key and a value. The result of \text{procedure} is discarded.

\[
\text{dbm-map} \ (dbm <dbm>) \ \text{procedure}
\]

{dbm} For each key/value pair in the database \text{dbm}, \text{procedure} is called. Two arguments are passed to \text{procedure}—a key and a value. The result of \text{procedure} is accumulated to a list which is returned as a result of \text{dbm-map}.

12.28.4 Managing dbm database instance

Each dbm implementation has its own way to store the database. Legacy dbm uses two files, whose names are generated by adding \text{.dir} and \text{.pag} to the value of \text{path} slot. Fsdbm creates a directory under \text{path}. If dbm database is backed up by some database server, \text{path} may be used only as a key to the database in the server. The following methods hide such variations and provides a convenient way to manage a database itself. You have to pass a class that implements a concrete dbm database to their first argument.

\[
\text{dbm-db-exists?} \ \text{class name}
\]

{dbm} Returns \#t if a database of class \text{class} specified by \text{name} exists.

\[
;\text{Returns } \#t \text{ if testdb.dir and testdb.pag exist}
\]

\[
\text{dbm-db-REMOVE} \ \text{class name}
\]

{dbm} Removes an entire database of class \text{class} specified by \text{name}.

\[
\text{dbm-db-COPY} \ \text{class from to}
\]

{dbm} Copy a database of class \text{class} specified by \text{from} to \text{to}. The integrity of \text{from} is guaranteed if the \text{class}'s dbm implementation supports locking (i.e. you won’t get a corrupted database even if some other process is trying to write to \text{from} during copy). If the destination database \text{to} exists, its content is destroyed. If this function is interrupted, whether \text{to} is left in incomplete state or not depends on the dbm implementation. The implementation usually tries its best to provide transactional behavior, that is, to recover original \text{to} when the copy...
fails. However, for the robust operations the caller have to check the state of to if `dbm-db-copy` fails.

```scheme
(dbm-db-copy <gdbm> "testdb.dbm" "backup.dbm")
```

dbm-db-move class from to  
{dbm} Moves or renames a database of class `class` specified by `from` to `to`. Like `dbm-db-copy`, the database integrity is guaranteed as far as class’s dbm implementation supports locking. If the destination database `to` exists, its content is destroyed.

### 12.28.5 Dumping and restoring dbm database

Most dbm implementations use some kind of binary format, and some of them are architecture dependent. That makes it difficult to pass around dbm databases between different machines. A safe way is to write out the content of a dbm database into some portable format on the source machine, and rebuild another dbm database from it on the destination machine.

The operation is so common that Gauche provides convenience scripts that does the job. They are installed into the standard Gauche library directory, so it can be invoked by `gosh <scriptname>`.

To write out the content of a dbm database named by `dbm-name`, you can use `dbm/dump` script:

```bash
$ gosh dbm/dump [-o outfile][-t type] dbm-name
```

The `outfile` argument names the output file. If omitted, the output is written out to stdout. The `type` argument specifies the implementation type of the dbm database; e.g. `gdbm` or `fsdbm`. The program calls `dbm-type->class` (see Section 12.28.1 [Opening and closing a dbm database], page 872) on the `type` argument to load the necessary dbm implementation.

The dumped format is simply a series of S-expressions, each of which is a dotted pair of string key and string value. Character encodings are assumed to be the same as `gosh`'s native character encoding.

The dumped output may contain S-expressions other than dotted pair of strings to include meta information. For now, programs that deals with dumped output should just ignore S-expressions other than dotted pairs.

To read back the dumped dbm format, you can use `dbm/restore` script:

```bash
$ gosh dbm/restore [-i infile][-t type] dbm-name
```

The `infile` argument names the dumped file to be read. If omitted, it reads from stdin. The `type` argument specifies the dbm type, as in `dbm/dump` script. The `dbm-name` argument names the dbm database; if the database already exists, its content is cleared, so be careful.

### 12.28.6 Writing a dbm implementation

When you write an extension module that behaves like a persistent hashtable, it is a good idea to adapt it to the dbm interface, so that the application can use the module in a generic way.

The minimum procedures to conform the dbm interface are as follow:

- Define a metaclass `<foo-meta>`. It doesn’t need to inherit anything except `<class>`.
- Define a dbm class `<foo>` that inherits `<dbm>` and whose metaclass is `<foo-meta>`.
- Define methods for `dbm-open`, `dbm-close`, `dbm-put!`, `dbm-get`, `dbm-exists`, `dbm-delete!`, `dbm-fold`, `dbm-closed?`, specialized for `<foo>`. (The case of `dbm-open` for `<foo-meta>` is handled automatically, so you don’t need to define it unless you want something special). Also note that the specialized `dbm-open` must call `next-method` in it to set up dbm base class internals.
- Define methods for `dbm-db-exists?` and `dbm-db-remove` on `<foo-meta>`.
Besides above, you may define the following methods.

- **Methods for `dbm-for-each` and `dbm-map`**. If you don't define them, a generic implementation by `dbm-fold` is used. There may be an implementation specific way which is more efficient.

- **Methods for `dbm-db-copy` and `dbm-db-move`**. If you don't define them, a fallback method opens the specified databases and copies elements one by one, and removes the original if the method is `dbm-db-move`. Note that the fallback method is not only inefficient, but also it may not copy any implementation-specific meta information. It is highly recommended for the `dbm` implementation to provide these methods as well.

It is generally recommended to name the implementation module as `dbm.foo`, and the class of the implementation as `<foo>`. With this convention it is easier to write an application that dynamically loads and uses `dbm` implementation specified at runtime.

### 12.29 dbm.fsdbm - File-system dbm

**dbm.fsdbm**

Implements fsdbm. Extends `dbm`.

**<fsdbm>**

{<dbm.fsdbm>} Fsdbm is a `dbm` implementation that directly uses the filesystem. Basically, it uses file names for keys, and file content for values. Unlike other `dbm` implementations, this doesn’t depend on external libraries—it is pure Scheme implementation—so it is always available, while other `dbm` implementations may not. Obviously, it is not suitable for the database that has lots of entries, or has entries deleted and added very frequently. The advantage is when the number of entries are relatively small, and the values are relatively large while keys are small. The database name given to `<fsdbm>` instance is used as a directory name that stores the data. The data files are stored in subdirectories under path of `fsdbm` instance, hashed by the key. Non-alphanumeric characters in the key is encoded like _3a for ':'_, for example. If a key is too long to be a file name, it is chopped to chunks, and each chunk but the last one is used as a directory name. Note that a long key name may still cause a problem, for example, some of old `tar` command can’t deal with pathnames (not each pathname components, but the entire pathname) longer than 256 characters.

Fsdbm implements all of the `dbm` protocol (see Section 12.28 [Generic DBM interface], page 871). It doesn’t have any fsdbm-specific procedures.

### 12.30 dbm.gdbm - GDBM interface

**dbm.gdbm**

Provides interface to the gdbm library. Extends `dbm`.

**<gdbm>**

{<dbm.gdbm>} Inherits `<dbm>`. Provides an implementation for GDBM library. This module is only installed when your system already has GDBM (1.8.0 is preferred, but works with older 1.7.x with some limitations).

- **sync**  
  [Instance Variable of `<gdbm>`]
- **nolock**  
  [Instance Variable of `<gdbm>`]
- **bsize**  
  [Instance Variable of `<gdbm>`]

Besides the unified DBM interface (see Section 12.28 [Generic DBM interface], page 871), this module provides the following low-level functions that provides direct access to the gdbm API. See gdbm manual for details of these APIs.
gdbm-open path :optional size rwmode fmode error-callback
{dbm.gdbm}

GDBM_READER
{dbm.gdbm}

GDBM_WRITER
{dbm.gdbm}

GDBM_WRCREAT
{dbm.gdbm}

GDBM_NEWDB
{dbm.gdbm}

GDBM_FAST
{dbm.gdbm}

GDBM_SYNC
{dbm.gdbm}

GDBM_NOLOCK
{dbm.gdbm}

gdbm-close gdbm-object
{dbm.gdbm}

gdbm-closed? gdbm-object
{dbm.gdbm}

gdbm-store key value :optional flag
{dbm.gdbm}

GDBM_INSERT
{dbm.gdbm}

GDBM_REPLACE
{dbm.gdbm}

gdbm-fetch gdbm-object key
{dbm.gdbm}

gdbm-delete gdbm-object key
{dbm.gdbm}

gdbm-firstkey gdbm-object
{dbm.gdbm}

gdbm-nextkey gdbm-object key
{dbm.gdbm}

gdbm-reorganize gdbm-object
{dbm.gdbm}

gdbm-sync gdbm-object
{dbm.gdbm}

gdbm-exists? gdbm-object key
{dbm.gdbm}
Chapter 12: Library modules - Utilities

12.31 dbm.ndbm - NDBM interface

**dbm.ndbm**

Provides interface to the 'new' dbm library, a.k.a. ndbm. Extends dbm.

**<ndbm>**

{dbm.ndbm} Inherits <dbm>. Provides an implementation for NDBM library. This module is only installed when your system already has NDBM.

Besides the unified DBM interface (see Section 12.28 [Generic DBM interface], page 871), this module provides the following low-level functions that provides direct access to the ndbm API. See ndbm manual for details of these APIs.

**ndbm-open** path flags mode

{dbm.ndbm}

**ndbm-close** ndbm-object

{dbm.ndbm}

**ndbm-closed?** ndbm-object

{dbm.ndbm}

**ndbm-store** ndbm-object key content :optional flag

{dbm.ndbm}

**ndbm-fetch** ndbm-object key

{dbm.ndbm}

**ndbm-delete** ndbm-object key

{dbm.ndbm}
Chapter 12: Library modules - Utilities

12.32 dbm.odbm - Original DBM interface

dbm.odbm

Provides interface to the legacy dbm library. Extends dbm.

<odbm>

{dbm.odbm} Inherits <dbm>. Provides an implementation for legacy DBM library. This module is only installed when your system already has DBM.

The biggest limitation of the legacy DBM is that you can only open one database at a time. You can create a multiple <odbm> instances, but you can open at most one of it at a time, or you’ll get an error.

Besides the unified DBM interface (see Section 12.28 [Generic DBM interface], page 871), this module provides the following low-level functions that provides direct access to the dbm API. See dbm manual for details of these APIs.

odbm-init path
{dbm.odbm}

odbm-close
{dbm.odbm}

odbm-store key value
{dbm.odbm}

odbm-fetch key
{dbm.odbm}

odbm-delete key
{dbm.odbm}

odbm-firstkey
{dbm.odbm}

odbm-nextkey key
{dbm.odbm}
Chapter 12: Library modules - Utilities

12.33 file.filter - Filtering file content

file.filter
This module provides utilities for a common pattern in filter-type commands, that is, to take an input, to process the content, and to write the result. The common occurring pattern is:

- Input may be a specified file, or an input port (the current input port by default).
- Output may be a specified file, or an output port (the current output port by default).
- Output may be a temporary file, which will be renamed upon completion of the processing.
- Output file may be removed when an error occurs in the processing.

file-filter proc :key input output temporary-file keep-output? [Function]
rename-hook
{file.filter} Calls proc with two arguments, an input port and an output port. Returns the result(s) of proc. The input port and output port are chosen depending on the keyword arguments.

input
The argument must be either an input port or a string that specifies a file name. If it’s an input port, it is passed to proc as is. If it’s a string, the named file is opened for input and the resulting port is passed to proc, and the port is closed when proc returns. If this argument is omitted, the current input port is passed.

output
The argument must be either an output port or a string that specifies a file name. If it’s an output port, it is passed to proc as is. If it’s a string, the named file is opened for output (unless temporary-file is given, in that case a temporary file is opened instead), and the resulting port is passed to proc. This port is closed when proc returns. If this argument is omitted, the current output port is passed.

temporary-file
The value must be a boolean or a string. If a non-false value is given, and output is a file, then a fresh temporary file is created and opened for output and passed to proc. When proc returns normally, the file is renamed to the name given to output keyword argument.

If #t is given, a temporary file name is generated based on the name of the output file. If a string file name is given to this argument, the name is used for sys-mkstemp.

If the given file name begins with characters except "/", "./" or ".../", the directory of the file name given to output argument is attached before it.

The default value is #f (do not use a temporary file).

keep-output?
If a true value is given, the output is not deleted even when proc signals an error. By default, the output (or the temporary file when temporary-file is given) will be deleted on error.

leave-unchanged
When a temporary file is used, and a true value is given to this argument, the existing output file is left intact when the generated output in the temporary file exactly matches the original content of the output file. It is useful if touching output file may trigger some actions (e.g. by make) and you want to avoid invoking unnecessary actions. The default value is #f (always replace the output).
file-filter-fold proc seed :key reader input output temporary-file
keep-output? rename-hook
{file.filter} A convenience wrapper of file-filter. Call proc for each item read from input by reader (read-line by default). The argument proc receives is the item, the seed value and the output port; proc can emit the output, as well as returning some value that is passed along as the seed value. Other keyword arguments are passed to file-filter.
For example, the following code reads each line from file.txt and displays lines matching #/regexp/ with line numbers.
(file-filer-fold
  (~[line nc out]
   (when (#/regexp/ line) (format out "~3d: "a\n" nc line))
  (+ nc 1))
 1 :input "file.txt")

file-filter-map proc :key reader input output temporary-file
keep-output? rename-hook
file-filter-for-each proc :key reader input output temporary-file
keep-output? rename-hook
{file.filter} Utilities similar to file-filter-fold, like map and for-each to fold.
The procedure proc is called with two arguments, an item read from the input and an output port. The results of proc are collected as a list and returned by file-filter-map, and discarded by file-filter-for-each.
The meaning of keyword arguments are the same as file-filter-fold.

12.34 file.util - Filesystem utilities

Provides convenient utility functions handling files and directories. Those functions are built on top of the primitive system procedures described in Section 6.24.4 [Filesystems], page 296.
Many procedures in this module takes a keyword argument follow-link?, which specifies the behavior when the procedure sees a symbolic link. If true value is given to follow-link? (which is the default), the procedure operates on the file referenced by the link; if false is given, it operates on the link itself.

Note on the naming convention: Some Scheme implementations "create" new directories and files, while the others "make" them. Some implementations "delete" them, while the others "remove" them. It seems that both conventions are equally popular. So Gauche provides both.

12.34.1 Directory utilities

current-directory :optional new-directory
{file.util} When called with no argument, this returns the pathname of the current working directory. When called with a string argument new-directory, this sets the current working directory of the process to it. If the process can’t change directory to new-directory, an error is signaled.
This function is in ChezScheme, MzScheme and some other Scheme implementations.
SRFI-170 defines current-directory without arguments, to return the current working directory.

home-directory :optional user
{file.util} Returns the home directory of the given user, which may be a string user name or an integer user id. If user is omitted, the current user is assumed. If the given user cannot be found, or the home directory of the user cannot be determined, #f is returned.
On Windows native platforms, this function is only supported to query the current user’s directory.

**directory-list path :key children? add-path? filter filter-add-path?**

{file.util} Returns a list of entries in the directory path. The result is sorted by dictionary order.

By default, only the basename (the last component) of the entries returned. If add-path? is given and true, path is appended to each entry. If children? is given and true, "." and ". ." are excluded from the result.

If filter is given, it must be a predicate that takes one argument. It is called on every element of the directory entry, and only the entries on which filter returns true are included in the result. The argument passed to filter is a basename of the directory entry by default, but when filter-add-path? is true, path is appended to the entry.

If path is not a directory, an error is signaled.

\[(directory-list "test")\]
\[⇒ ("." ".." "test.scm" "test.scm~")\]

\[(directory-list "test" :add-path? #t)\]
\[⇒ ("test/." "test/.." "test/test.scm" "test/test.scm~")\]

\[(directory-list "test" :children? #t)\]
\[⇒ ("test.scm" "test.scm~")\]

\[(directory-list "test" :children? #t :add-path? #t :filter (lambda (e) (not (string-suffix? "~" e))))\]
\[⇒ ("test/test.scm")\]

**directory-list2 path :key children? add-path? filter follow-link?**

{file.util} Like directory-list, but returns two values; the first one is a list of subdirectories, and the second one is a list of the rest. The keyword arguments children?, add-path? and filter are the same as directory-list.

Giving false value to follow-link? makes directory-list2 not follow the symbolic links; if the path contains a symlink to a directory, it will be included in the first list if follow-link? is omitted or true, while it will be in the second list if follow-link? is false.

**directory-fold path proc seed :key lister follow-link?**

{file.util} A fundamental directory traverser. Conceptually it works as follows, in recursive way.

- If path is not a directory, calls (proc path seed) and returns the result.
- If path is a directory, calls (lister path seed). The procedure lister is expected to return two values: a list of pathnames, and the next seed value. Then directory-fold is called on each returned pathname, passing the returned seed value to the seed argument of the next call of directory-fold. Returns the result of the last seed value.

The default procedure of lister is just a call to directory-list, as follows.

\[(lambda (path seed)
  (values (directory-list path :add-path? #t :children? #t)
    seed))\]

Note that lister shouldn’t return the given path itself (". ") nor the parent directory (" .. "), or the recursion wouldn’t terminate. Also note lister is expected to return a
path accessible from the current directory, i.e. if \textit{path} is "/usr/lib/foo" and it contains "libfoo.a" and "libfoo.so", \textit{lister} should return '("/usr/lib/foo/libfoo.a" "/usr/lib/foo/libfoo.so")'.

The keyword argument \textit{follow-link?} is used to determine whether \textit{lister} should be called on a symbolic link pointing to a directory. When \textit{follow-link?} is true (default), \textit{lister} is called with the symbolic link if it points to a directory. When \textit{follow-link?} is false, \textit{proc} is not called.

The following example returns a list of pathnames of the emacs backup files (whose name ends with "~") under the given path.

\begin{verbatim}
(use srfi.13) ;; for string-suffix?
(directory-fold path
  (lambda (entry result)
    (if (string-suffix? "~" entry)
        (cons entry result)
        result))
  '()))
\end{verbatim}

The following example lists all the files and directories under the given pathname. Note the use of \textit{lister} argument to include the directory path itself in the result.

\begin{verbatim}
(directory-fold path cons '())
:lister (lambda (path seed)
  (values (directory-list path :add-path? #t :children? #t)
          (cons path seed))))
\end{verbatim}

\begin{function}
\textbf{make-directory* name :optional perm}
\end{function}

\begin{function}
\textbf{create-directory* name :optional perm}
\end{function}

\{file.util\} Creates a directory \textit{name}. If the intermediate path to the directory doesn’t exist, they are also created (like \texttt{mkdir -p} command on Unix). If the directory \textit{name} already exist, these procedure does nothing. \textit{Perm} specifies the integer flag for permission bits of the directory.

\begin{function}
\textbf{remove-directory* name}
\end{function}

\begin{function}
\textbf{delete-directory* name}
\end{function}

\{file.util\} Deletes directory \textit{name} and its content recursively (like \texttt{rm -r} command on Unix). Symbolic links are not followed.

\begin{function}
\textbf{copy-directory* src dst :key if-exists backup-suffix safe keep-timestamp keep-mode follow-link?}
\end{function}

\{file.util\} If \textit{src} is a regular file, copies its content to \textit{dst}, just like \texttt{copy-file} does. If \textit{src} is a directory, recursively descends it and copy the file tree to \textit{dst}. Basically it mimics the behavior of \texttt{cp -r} command.

If there’s any symbolic links under \textit{src}, the link itself is copied instead of the file pointed to by it, unless a true value is given to the \textit{follow-link?} keyword argument, i.e. the default value of \textit{follow-link?} is \texttt{#f}. (Note that this is opposite to the \texttt{copy-file}, in which \textit{follow-link?} is true by default.)

The meanings of the other keyword arguments are the same as \texttt{copy-file}. See the entry of \texttt{copy-file} for the details.

\begin{function}
\textbf{create-directory-tree dir spec}
\end{function}

\{file.util\} Creates a directory tree under \textit{dir} according to \textit{spec}. This procedure is useful to set up certain directory hierarchy at once.

The \textit{spec} argument is an S-expression with the following structure:

\begin{verbatim}
<spec> : <name> ; empty file
\end{verbatim}
Chapter 12: Library modules - Utilities

| (<name> <option> ...) ; empty file  
| (<name> <option> ... <string>) ; file with content  
| (<name> <option> ... <procedure>) ; file with generated content  
| (<name> <option> ... (<spec> ...)) ; directory

<name> : string or symbol

<option> ... : keyword-value alternating list

With the first and second form of spec, an empty file is created with the given name. With the third form of spec, the string becomes the content of the file.

With the fourth form of spec, the procedure is called with the pathname as an argument, and output to the current output port within the procedure is written to the created file. The pathname is relative to the dir argument. At the time the procedure is called, its parent directory is already created.

The last form of spec creates a named directory, then creates its children recursively according to the specs.

With options you can control attributes of created files/directories. Currently the following options are recognized.

:mode mode
  Takes integer as permission mode bits.

:owner uid
:group gid
  Takes integer uid/gid of the owner/group of the file/directory. Calling process may need special privilege to change the owner and/or group.

:symlink path
  This is only valid for file spec, and it causes create-directory-tree to create a named symbolic link whose content is path.

check-directory-tree dir spec  {file.util}
  Checks if a directory hierarchy according to spec exists under dir. Returns #t if it exists, or #f otherwise.

  The format of spec is the same as create-directory-tree described above.

  If spec contains options, the attributes of existing files/directories are also checked if they match the given options.

12.34.2 Pathname utilities

build-path base-path component ...
  {file.util} Appends pathname components component to the base-path. Component can be a symbol up or same; in Unix, they are synonym to "." and "..". This API is taken from MzScheme.

absolute-path? path  {file.util}
  Returns #t if path is absolute or relative, respectively.

relative-path? path  {file.util}

expand-path path  {file.util}
  Expands tilda-notation of path if it contains one. Otherwise, path is returned. This function does not check if path exists and/or readable.
**resolve-path path**  
{file.util} Expands path like `expand-path`, then resolve symbolic links for every components of the path. If path does not exist, or contains dangling link, or contains unreadable directory, an error is signaled.

**simplify-path path**  
{file.util} Remove 'up' ("..") components and 'same' (".") components from path as much as possible. This function does not access the filesystem.

**decompose-path path**  
{file.util} Returns three values; the directory part of path, the basename without extension of path, and the extension of path. If the pathname doesn’t have an extension, the third value is #f. If the path name ends with a directory separator, the second and third values are #f. (Note: This treatment of the trailing directory separator differs from sys-dirname/sys-basename; those follow popular shell’s convention, which ignores trailing slashes.)

```
(decompose-path "/foo/bar/baz.scm")
⇒ "/foo/bar", "baz", "scm"
(decompose-path "/foo/bar/baz")
⇒ "/foo/bar", "baz", #f

(decompose-path "baz.scm")
⇒ ".", "baz", "scm"
(decompose-path "/baz.scm")
⇒ "/", "baz", "scm"
```

**path-extension path**  
**path-sans-extension path**  
{file.util} Returns an extension of path, and a pathname of path without extension, respectively. If path doesn’t have an extension, #f and path is returned respectively.

```
(path-extension "/foo/bar.c")  ⇒ "c"
(path-sans-extension "/foo/bar.c")  ⇒ "/foo/bar"

(path-extension "/foo/bar")  ⇒ #f
(path-sans-extension "/foo/bar")  ⇒ "/foo/bar"
```

**path-swap-extension path newext**  
{file.util} Returns a pathname in which the extension of path is replaced by newext. If path doesn’t have an extension, "." and newext is appended to path. If newext is #f, it returns path without extension.

```
(path-swap-extension "/foo/bar.c" "o")  ⇒ "/foo/bar.o"
(path-swap-extension "/foo/bar.c" "")  ⇒ "/foo/bar."
(path-swap-extension "/foo/bar.c" #f)  ⇒ "/foo/bar"
```
(path-swap-extension "/foo/bar" "o")  ⇒ "/foo/bar.o"
(path-swap-extension "/foo/bar" "")  ⇒ "/foo/bar."
(path-swap-extension "/foo/bar" #f)  ⇒ "/foo/bar"

(find-file-in-paths name :key paths pred extensions)  [Function]
{file.util} Looks for a file that has name name in the given list of pathnames paths and that satisfies a predicate pred. If found, the absolute pathname of the file is returned. Otherwise, #f is returned.

If name is an absolute path, only the existence of name and whether it satisfies pred are checked.

The default value of paths is taken from the environment variable PATH, and the default value of pred is file-is-executable? (see Section 12.34.3 [File attribute utilities], page 886). That is, find-file-in-paths searches the named executable file in the command search paths by default.

(find-file-in-paths "ls")  ⇒ "/bin/ls"

;; example of searching user preference file of my application
(find-file-in-paths "userpref"
  :paths `(,(expand-path "~/.myapp"
         "/usr/local/share/myapp"
         "/usr/share/myapp")
  :pred  file-is-readable?)

The extensions keyword argument may list alternative extensions added to name. For example, the following example searches not only notepad, but also notepad.exe and notepad.com, in the PATH. If an alternate name is found, the returned pathname contains the extension.

(find-file-in-paths "notepad" :extensions '("exe" "com"))

For each path, the name and the alternative names are checked in order. That is, if there are /bin/b.com and /usr/bin/b.exe and paths is "(/bin "/usr/bin")", you'll get /bin/b.com when you search b with extensions ("exe" "com").

null-device  [Function]
{file.util} Returns a name of the null device. On unix platforms (including cygwin) it returns "/dev/null", and on Windows native platforms (including mingw) it returns "NUL".

console-device  [Function]
{file.util} Returns a name of the console device. On unix platforms (including cygwin) it returns "/dev/tty", and on Windows native platforms (including mingw) it returns "CON".

This function does not guarantee the device is actually available to the calling process.

12.34.3 File attribute utilities

file-type path :key follow-link?  [Function]
file-perm path :key follow-link?  [Function]
file-mode path :key follow-link?  [Function]
file-ino path :key follow-link?  [Function]
file-rdev path :key follow-link?  [Function]
file-dev path :key follow-link?  [Function]
file-nlink path :key follow-link?  [Function]
file-uid path :key follow-link?  [Function]
file-gid path :key follow-link?  [Function]
file-size path :key follow-link?
file-atime path :key follow-link?
file-mtime path :key follow-link?
file-ctime path :key follow-link?
{file.util} These functions return the attribute of file/directory specified by path. The attribute name corresponds to the slot name of <sys-stat> class (see Section 6.24.4.4 [File stats], page 301). If the named path doesn’t exist, #f is returned.

If path is a symbolic link, these functions queries the attributes of the file pointed by the link, unless an optional argument follow-link? is given and false.

MzScheme and Chicken have file-size. Chicken also has file-modification-time, which is file-mtime.

file-is-readable? path
file-is-writable? path
file-is-executable? path
{file.util} Returns #t if path exists and readable/writable/executable by the current effective user, respectively. This API is taken from STk.

file-is-symlink? path
{file.util} Returns #t if path exists and a symbolic link. See also file-is-regular? and file-is-directory? in Section 6.24.4.4 [File stats], page 301.

file-eq? path1 path2
file-eqv? path1 path2
file-equal? path1 path2
{file.util} Compares two files specified by path1 and path2. file-eq? and file-eqv? checks if path1 and path2 refers to the identical file, that is, whether they are on the same device and have the identical inode number. The only difference is when the last component of path1 and/or path2 is a symbolic link. file-eq? doesn’t resolve the link (so compares the links themselves) while file-eqv? resolves the link and compares the files referred by the link(s).

de-equal? compares path1 and path2 considering their content, that is, when two are not the identical file in the sense of file-eqv?, file-equal? compares their content and returns #t if all the bytes match.

The behavior of file-equal? is undefined when path1 and path2 are both directories. Later, it may be extended to scan the directory contents.

file-mtime=? f1 f2
file-mtime<? f1 f2
file-mtime<=? f1 f2
file-mtime>? f1 f2
file-mtime=>? f1 f2
{file.util} Compares file modification time stamps. There are a bunch of methods defined, so each argument can be either one of the followings.

- String pathname. The mtime of the specified path is used.
- <sys-stat> object (see Section 6.24.4.4 [File stats], page 301). The mtime is taken from the stat structure.
- <time> object. The time is used as the mtime.
- Number. It is considered as the number of seconds since Unix Epoch, and used as mtime.

;; compare "foo.c" is newer than "foo.o"
(file-mtime>? "foo.c" "foo.o")
;; see if "foo.log" is updated within last 24 hours
(file-mtime>? "foo.c" (- (sys-time) 86400))

file-ctime=? f1 f2  [Generic Function]
file-atime=? f1 f2  [Generic Function]

{file.util} Same as file-mtime=?, except these checks file’s change time and access time,
respectively. All the variants of <, <=, >, >= are also defined.

12.34.4 File operations

touch-file  path :key (time #f) (type #f) (create #t)  [Function]
touch-files paths :key (time #f) (type #f) (create #t)  [Function]

{file.util} Updates timestamp of path, or each path in the list paths, to the current time.
If the specified path doesn’t exist, a new file with size zero is created, unless the keyword
argument create is #f.

If the keyword argument time is given and not #f, it must be a nonnegative real number. It
is used as the timestamp value instead of the current time.

The keyword argument type can be #f (default), a symbol atime or mtime. If it is a symbol,
only the access time or modification time is updated.

Note: touch-files processes one file at a time, so the timestamp of each file may not be
exactly the same.

These procedures are built on top of the system call sys-utime (see Section 6.24.4.4 [File
stats], page 301).

copy-file src dst :key if-exists backup-suffix safe keep-timestamp
  keep-mode follow-link?  [Function]

{file.util} Copies file from src to dst. The source file src must exist. The behavior when
the destination dst exists varies by the keyword argument if-exists;

:error         (Default) Signals an error when dst exists.
:supersede     Replaces dst to the copy of src.
:backup        Keeps dst by renaming it.
:append         Append the src’s content to the end of dst.
#f             Doesn’t copy and returns #f when dst exists.

Copy-file returns #t after completion.

If src is a symbolic link, copy-file follows the symlink and copies the actual content by
default. An error is raised if src is a dangling symlink.

Giving #f to the keyword argument follow-link? makes copy-file to copy the link itself. It
is possible that src is a dangling symlink in this case.

If if-exists is :backup, the keyword argument backup-suffix specifies the suffix attached to
the dst to be renamed. The default value is ".orig".

By default, copy-file starts copying to dst directly. However, if the keyword argument safe
is a true value, it copies the file to a temporary file in the same directory of dst, then renames
it to dst when copy is completed. (When safe is true and if-exists is :append, we first copy
the content of dst to a temporary file if dst exists, appends the content of src, then renames
the result to dst). If copy is interrupted for some reason, the filesystem is "rolled back"
properly.
If the keyword argument *keep-timestamp* is true, *copy-file* sets the destination’s timestamp to the same as the source’s timestamp after copying.

If the keyword argument *keep-mode* is true, the destination file’s permission bits are set to the same as the source file’s. If it is false (default), the destination file’s permission remains the same if the destination already exists and the *safe* argument is false, otherwise it becomes #o666 masked by umask settings.

**move-file**

```scheme
src dst :key if-exists backup-suffix
{file.util}
```

Moves file *src* to *dst*. The source *src* must exist. The behavior when *dst* exists varies by the keyword argument *if-exists*, as follows.

- :error (Default) Signals an error when *dst* exists.
- :supersede Replaces *dst* by *src*.
- :backup Keeps *dst* by renaming it.
- #f Doesn’t move and returns #f when *dst* exists.

**Move-file** returns #t after completion.

If *if-exists* is :backup, the keyword argument *backup-suffix* specifies the suffix attached to the *dst* to be renamed. The default value is ".orig".

The file *src* and *dst* can be on the different filesystem. In such a case, **move-file** first copies *src* to the temporary file on the same directory as *dst*, then renames it to *dst*, then removes *src*.

**remove-file**

```scheme
filename
{R7RS file} {file.util}
```

Removes the named file. An error is signalled if *filename* does not exist, is a directory, or cannot be deleted with other reasons such as permissions. R7RS defines **delete-file**.

Compare with sys-unlink (see Section 6.24.4.2 [Directory manipulation], page 299), which doesn’t raise an error when the named file doesn’t exist.

**remove-files**

```scheme
paths
{file.util}
```

Removes each path in a list *paths*. If the path is a file, it is unlinked. If it is a directory, its contents are recursively removed by **remove-directory**.*. If the path doesn’t exist, it is simply ignored.

**delete-files** is just an alias of **remove-files**.

**file->string**

```scheme
filename options ...
{file.util}
```

Convenience procedures to read from a file *filename*. They first open the named file, then call port->string, port->list, port->string-list and port->sexp-list on the opened file, respectively. (see Section 6.21.7.4 [Input utility functions], page 274). The file is closed if all the content is read or an error is signaled during reading.

Those procedures take the same keyword arguments as **call-with-input-file**. When the named file doesn’t exist, the behavior depends on *if-does-not-exist* keyword argument—an error is signaled if it is :error, and #f is returned if the argument is #f.
string->file filename string options . . .  
list->file writer filename lis options . . .
string-list->file filename lis options . . .
sexp-list->file filename lis options . . .

{file.util} Opposite of file->string etc. They are convenient to quickly write out things into a file.

NB: The name string->file etc. might suggest they would take the object to be written as the first argument. We decided to put filename first, since in the situations where these procedures are used, it is more likely that one want to write literal data, which would be bigger than the filename itself.

The options part is passed to call-with-output-file as is. For example, the following code appends the text when foo.txt already exists:

```
(string->file "foo.txt" "New text to append\n" :if-exists :append)
```

The list->file takes writer argument, which is a procedure that receives two arguments, an element from the list lis, and an output port. It should write out the element to the port in a suitable way. The string-list->file and sexp-list->file are specialized versions of list->file, where string-list->file uses (^[s p] (display s p) (newline p)) as writer, and sexp-list->file uses (^[s p] (write s p) (newline p)) as writer.

12.34.5 Temporary files and directories

temporary-directory  
{file.util} A parameter that keeps the name of the directory that can be used to create a temporary files. The default value is the one returned from sys-tmpdir (see Section 6.24.4.3 [Pathnames], page 300). The difference of sys-tmpdir is that, since this is a parameter, it can be overridden by application during execution. Libraries are recommended to use this instead of sys-tmpdir for greater flexibility.

call-with-temporary-file proc :key directory prefix  
{file.util} Creates a temporary file with a unique name and opens it for output, then calls proc with the output port and the temporary file's name. The temporary file is removed after either proc returns or raises an uncaught error. Returns the value(s) proc returns.

The temporary file is created in the directory directory, with the name prefix followed by several random alphanumeric characters. When omitted, the value of (temporary-directory) is used for directory, and "gtemp" for prefix.

The name passed to proc consists of directory and the file’s name. So whether the name is absolute or relative pathname depends on the value of directory.

```
(call-with-temporary-file (^[_ name] name) ⇒ Something like "/tmp/gtemp4dSpMh"
```

You can keep the output file by renaming it in proc. But if doing so, make sure to specify directory so that the temporary file is created in the same directory as the final output; rename may not work across filesystems. If you anticipate your code runs on Windows as well, make sure to close the output port before renaming. Windows does not allow you to rename an opened file.

Internally, it calls sys-mkstemp to create a unique file. See Section 6.24.4.2 [Directory manipulation], page 299, for the details.

call-with-temporary-directory proc :key directory prefix  
{file.util} Creates a temporary directory with unique name, then calls proc with the name. The temporary directory and its contents are removed after either proc returns or raises an uncaught error. Returns the value(s) proc returns.
The temporary directory is created in the directory directory, with the name prefix followed by several random alphanumeric characters. When omitted, the value of (temporary-directory) is used for directory, and "gtemp" for prefix.

The name passed to proc consists of directory and the directory name. So whether the name is absolute or relative pathname depends on the value of directory.

Internally, it calls sys-mkdtemp to create a unique file. See Section 6.24.4.2 [Directory manipulation], page 299, for the details.

### 12.34.6 Lock files

Exclusivity of creating files or directories is often used for inter-process locking. The following procedure provides a packaged interface for it.

```scheme
with-lock-file lock-name thunk :key type retry-interval retry-limit
    secondary-lock-name retry2-interval retry2-limit perms abandon-timeout
{file.util} Exclusively creates a file or a directory (lock file) with lock-name, then executes thunk. After thunk returns, or an error is thrown in it, the lock file is removed. When thunk returns normally, its return values become the return values of with-lock-file.

If the lock file already exists, with-lock-file waits and retries getting the lock until timeout reaches. It can be configured by the keyword arguments.

There's a chance that with-lock-file leaves the lock file when it gets a serious error situation and doesn't have the opportunity to clean up. You can allow with-lock-file to steal the lock if its timestamp is too old; say, if you know that the applications usually locks just for seconds, and you find the lock file is 10 minutes old, then it's likely that the previous process was terminated abruptly and couldn't clean it up. You can also configure this behavior by the keyword arguments.

Internally, two lock files are used to implement this stealing behavior safely. The creation and removal of the primary lock file (named by lock-name argument) are guarded by the secondary lock file (named by secondary-lock-file argument, defaulted by .2 suffix attached to lock-name). The secondary lock prevents more than one process steals the same primary lock file simultaneously.

The secondary lock is acquired for a very short period so there's much less chance to be left behind by abnormal terminations. If it happens, however, we just give up; we don't steal the secondary lock.

If with-lock-file couldn't get a lock before timeout, a <lock-file-failure> condition is thrown.

Here's a list of keyword arguments.

- **type**
  
  It can be either one of the symbols file or directory.
  
  If it is file, we use a lock file, relying on the _O_EXCL exclusive creation flag of open(2). This is the default value. It works for most platforms; however, some NFS implementation may not implement the exclusive semantics properly.
  
  If it is directory, we use a lock directory, relying on the atomicity of mkdir(2). It should work for any platforms, but it may be slower than file.

- **retry-interval**

- **retry-limit**

  Accepts a nonnegative real number that specifies either the interval to attempt to acquire the primary lock, or the maximum time we should keep retrying, respectively, in seconds. The default value is 1 second interval and 10 second limit. To prevent retrying, give 0 to retry-limit.
secondary-lock-name

The name of the secondary lock file (or directory). If omitted, lock-name with a suffix \texttt{.2} attached is used. Note: The secondary lock name must be agreed on all programs that locks the same (primary) lock file. I recommend to leave this to the default unless there’s a good reason to do otherwise.

retry2-interval
retry2-limit

Like retry-interval and retry-limit, but these specify interval and timeout for the secondary lock file. The possibility of secondary lock file collision is usually pretty low, so you would hardly need to tweak these. The default values are 1 second interval and 10 second limit.

perms

Specify the permission bitmask of the lock file or directory, in a nonnegative exact integer. The default is \texttt{#o644} for a lock file and \texttt{#o755} for a lock directory. Note that to control who can acquire/release/steal the lock, what matters is the permission of the directory in which the lock file/directory, not the permission of the lock file/directory itself.

abandon-timeout

Specifies the period in seconds in a nonnegative real number. If the primary lock file is older than that, with-lock-file steals the lock. To prevent stealing, give \texttt{#f} to this argument. The default value is 600 seconds.

<lock-file-failure> [Condition type]
{file.util} A condition indicating that with-lock-file couldn’t obtain the lock. Inherits <error>.

lock-file-name [Instance Variable of <lock-file-failure>]

The primary lock file name.

Gauche also provides OS-supported file locking feature, \texttt{fcntl} lock, via \texttt{gauche.fcntl} module. Whether you want to use \texttt{fcntl} lock or with-lock-file will depend on your application.

These are the advantages of the \texttt{fcntl} lock:

• The lock is removed when the process dies without explicitly unlocking it.
• You can directly lock the file you’re touching.
• You can lock a part of a file.
• You can have shared (read) and exclusive (write) locks.

In common situations, probably the most handy property is the first one; you don’t need to worry about leaving lock behind unexpected process termination.

However, there are a couple of shortcomings in \texttt{fcntl} locks.

• It is not guaranteed to work across different platforms, and/or NFS-mounted filesystems.
• The lock is per-process, per-file, and non-recursive. If you have a lock in a file, then calls a library that also locks the file, the lock always succeeds. Worse, if the library unlocks the file, the lock is completely removed, while the caller doesn’t know about it. It also means that, in order to prevent multiple threads in a process from accessing the same file, you have to use mutex along the \texttt{fcntl} lock.

Especially because of the second point, it is very difficult to use \texttt{fcntl} lock unless you have total control over and knowledge of the entire application. It is ok to use the \texttt{fcntl} lock by the application code to lock the application-specific file. Library developers have difficulty, however, to make sure any potential user of the library won’t try to lock the same file as the library tries to lock (usually it’s impossible).
12.35 math.const - Mathematical constants

math.const
This module defines several commonly-used mathematic constants.

- pi [Constant]
- π [Constant]
- 2π [Constant]
- π/2 [Constant]
- π/4 [Constant]
- π/180 [Constant]
- 1/π [Constant]
- 180/π [Constant]

{math.const} Bound to constants computed with π.

- e [Constant]

{math.const} Napier’s constant.

12.36 math.mt-random - Mersenne Twister Random number generator

math.mt-random
Provides a pseudo random number generator (RNG) based on "Mersenne Twister" algorithm developed by Makoto Matsumoto and Takuji Nishimura. It is fast, and has huge period of $2^{19937}-1$. See https://dl.acm.org/citation.cfm?id=272995, for details about the algorithm.

For typical use cases of random number generators, we recommend to use srfi.27 which is implemented on top of this module and provides portable API (see Section 11.8 [Sources of random bits], page 706). You should use this module directly only when you need functions that aren’t available through srfi.27.

<mersenne-twister>
{math.mt-random} A class to encapsulate the state of Mersenne Twister RNG. Each instance of this class has its own state, and can be used as an independent source of random bits if initialized by individual seed.

make-mersenne-twister :optional seed private? [Function]
{math.mt-random} Creates and returns a new <mersenne-twister> RNG instance.

If seed argument is given, it must be either nonnegative exact integer, u32vector, or #f. The key can be any length.

If seed is omitted or #f, the RNG is initialized to the fixed value, so that it generates the same sequences of random numbers. If you want different sequence for each instance, you have to give different seeds explicitly, or initialize each instance with different seeds using mt-random-set-seed! below.

By default, the created RNG is thread-safe; you can call the generator from multiple thread without worrying to break its internal state.
However, if you know you only use it in a single thread, or you use separate mutex and know the generator call is always protected, then you can give a true value to private? optional argument to avoid overhead of locking.

```
(define m (make-mersenne-twister (sys-time)))

(mt-random-real m) ⇒ 0.10284287848537865
(mt-random-real m) ⇒ 0.463277483488805
(mt-random-real m) ⇒ 0.8628500643709712
...
```

**mt-random-set-seed!** *mt seed*  
{math.mt-random} Sets random seed value *seed* to the Mersenne Twister RNG (MTRNG) *mt*. *Seed* can be an arbitrary positive exact integer, or arbitrary length of u32vector (see Section 11.3 [Homogeneous vectors], page 691).

Internally, MTRNG keeps its state in an array of 624 32-bit integers. If the given seed is small (e.g. fixnum), it can’t create enough variations. Particularly, if the seed is within 32bits, it can be computed by sampling a couple of result from MTRNG. If you want the random sequence harder to predict, prepare your own u32vector seed filled with high-entropy bits.

Note that Mersenne Twister is never intended for cryptography, you shouldn’t use it for security-sensitive purposes.

NB: Up to 0.9.9, when *seed* is a bignum, we roll our own way to fold it in 32bit integer and then called Mersenne-Twister’s initialization function. It loses the entropy, so we changed it and now all the bits in the bignum is used for the seed.

**mt-random-get-seed** *mt*  
{math.mt-random} Returns the last seed value used to initialize Mersenne Twister RNG *mt*. It is either an exact integer or u32vector. If *mt* has never been initialized, #<undef> is returned.

**mt-random-get-state** *mt*  
**mt-random-set-state!** *mt state*  
{math.mt-random} Retrieves and reinstalls the state of Mersenne Twister RNG *mt*. The state is represented by a u32vector of 625 elements. The state can be stored elsewhere, and then restored to an instance of <mersenne-twister> to continue to generate the pseudo random sequence.

**mt-random-real** *mt*  
**mt-random-real0** *mt*  
{math.mt-random} Returns a random real number between 0.0 and 1.0. 1.0 is not included in the range. Mt-random-real doesn’t include 0.0 either, while mt-random-real0 does. Excluding 0.0 is from the draft SRFI-27.

**mt-random-integer** *mt range*  
{math.mt-random} Returns a random exact positive integer between 0 and range-1. Range can be any positive exact integer.

**mt-random-fill-u32vector!** *mt u32vector*  
**mt-random-fill-f32vector!** *mt f32vector*  
**mt-random-fill-f64vector!** *mt f64vector*  
{math.mt-random} Fills the given uniform vector by the random numbers. For mt-random-fill-u32vector!, the elements are filled by exact positive integers between 0 and 2^32-1. For mt-random-fill-f32vector! and mt-random-fill-f64vector!, it is filled by an inexact real number between 0.0 and 1.0, exclusive.
If you need a bunch of random numbers at once, these are much faster than getting one by one.

### 12.37 math.prime - Prime numbers

#### math.prime

This module provides utilities related to prime numbers.

#### Sequence of prime numbers

*primes*  
{math.prime} An infinite lazy sequence of primes.

```lisp
;; show 10 prime numbers from 100-th one.
(take (drop *primes* 100) 10)  
⇒ (547 557 563 569 571 577 587 593 599 601)
```

**reset-primes**  
{math.prime} Once you take a very large prime out of *primes*, all primes before that has been calculated remains in memory, since the head of sequence is held in *primes*. Sometimes you know you need no more prime numbers and you wish those calculated ones to be garbage-collected. Calling reset-primes rebinds *primes* to unrealized lazy sequence, allowing the previously realized primes to be GCed.

**primes**  
{math.prime} Returns a fresh lazy sequence of primes. It is useful when you need certain primes in a short period of time—if you don’t keep a reference to the head of the returned sequence, it will be garbage collected after you’ve done with the primes. (Note that calculation of a prime number needs the sequence of primes from the beginning, so even if your code only keep a reference in the middle of the sequence, the entire sequence will be kept in the thunk within the lazy sequence—you have to release all references in order to make the sequence GCed.)

On the other hand, each sequence returned by primes are realized individually, duplicating calculation.

The rule of thumb is—if you use primes repeatedly throughout the program, just use *primes* and you’ll save calculation. If you need primes one-shot, call primes and abandon it and you’ll save space.

#### Testing primality

**small-prime? n**  
{math.prime} For relatively small positive integers (below *small-prime-bound*, to be specific), this procedure determines if the input is prime or not, quickly and deterministically. If n is on or above the bound, this procedure returns #f.

This can be used to quickly filter out known primes; it never returns #t on composite numbers (while it may return #f on large prime numbers). Miller-Rabin test below can tell if the input is composite for sure, but it may return #t on some composite numbers.

*small-prime-bound*  
{math.prime} For all positive integers below this value (slightly above 3.4e14 in the current implementation), small-prime? can determines whether it is a prime or not.

**miller-rabin-prime? n**  
{math.prime} Check if an exact integer n is a prime number, using probabilistic Miller-Rabin algorithm (n must be greater than 1). If this procedure returns #f, n is a composite number.
If this procedure returns #t, n is likely a prime, but there's a small probability that it is a false positive.

Note that if n is smaller than a certain number (*small-prime-bound*), the algorithm is deterministic; if it returns #t, n is certainly a prime.

If n is greater than or equal to *small-prime-bound*, we use a probabilistic test. We choosing random base integer to perform Miller-Rabin test up to 7 times by default. You can change the number of tests by the keyword argument num-tests. The error probability (to return #t for a composite number) is at most (expt 4 (- num-tests)).

For a probabilistic test, miller-rabin-prime? uses its own fixed random seed by default. We chose fixed seed so that the behavior can be reproducible. To change the random sequence, you can provide your own random integer generator to the random-integer keyword argument. It must be a procedure that takes a positive integer k and returns a random integer from 0 to k-1, including.

### Function

**bpsw-prime? n**

`{math.prime}` Check if an exact integer n is a prime number, using Baillie-PSW primality test ([http://www.trnicely.net/misc/bpsw.html](http://www.trnicely.net/misc/bpsw.html)). It is deterministic, and returns the definitive answer below 2^64 (around 1.8e19). For larger integers this can return #t on a composite number, although such number hasn't been found yet. This never returns #f on a prime number.

This is slower than Miller-Rabin but fast enough for casual use, so it is handy when you want a definitive answer below the above range.

### Factorization

**naive-factorize n :optional divisor-limit**

`{math.prime}` Factorize a positive exact integer n by trying to divide it with all primes up to (sqrt n). Returns a list of prime factors (each of which is equal to or greater than 2), smaller ones first.

```scheme
(naive-factorize 142857)  
⇒ (3 3 3 11 13 37)
```

Note that (naive-factorize 1) is ()

Although this is pretty naive method, this works well as far as any of n's factors are up to the order of around 1e7. For example, the following example runs in about 0.4sec on 2.4GHz Core2 machine. (The first time will take about 1.3sec to realize lazy prime sequences.)

```scheme
(naive-factorize 3644357367494986671013))  
⇒ (10670053 10670053 32010157)
```

Of course, if n includes any factors above that order, the performance becomes abysmal. So it is better to use this procedure below 1e14 or so.

Alternatively, you can give divisor-limit argument that specifies the upper bound of the prime number to be tried. If it is given, naive-factorize leaves a factor f as is if it can't be divided by any primes less than or equal to divisor-limit. So, the last element of the returned list may be composite number. This is handy to exclude trivial factors before applying more sophisticated factorizing algorithms.

```scheme
(naive-factorize 825877877739 1000)  
⇒ (3 43 6402154091)
```

;; whereas

```scheme
(naive-factorize 825877877739)  
⇒ (3 43 4591 1394501)
```

The procedure also memoizes the results on smaller n to make things faster.
**mc-factorize** \( n \)  

This one is capable to handle much larger range than **naive-factorize**, somewhere around \( 1e20 \) or so.

Since this method is probabilistic, the execution time may vary on the same \( n \). But it will always return the definitive results as far as every prime factor of \( n \) is smaller than \( 2^{64} \).

At this moment, if \( n \) contains a prime factor greater than \( 2^{64} \), this routine would keep trying factorizing it forever. Practical applications should have some means to interrupt the function and give it up after some time bounds. This will be addressed once we have deterministic primality test.

**Miscellaneous**

**jacobi** \( a \) \( n \)  
{math.prime} Calculates Jacobi symbol \((a/n)\) (http://en.wikipedia.org/wiki/Jacobi_symbol).

**totient** \( n \)  
{math.prime} Euler’s totient function of nonnegative integer \( n \).

The current implementation relies on **mc-factorize** above, so it may take very long if \( n \) contains large prime factors.

### 12.38 math.simplex - Simplex solver

**math.simplex**  
Implements the simplex algorithm to solve linear programming problems.

**simplex-solve** \( A \) \( b \) \( goal \) \( c \)  
{math.simplex} \( A \) is an \( m \times n \) array of numbers, \( b \) is a sequence of \( m \) numbers, \( c \) is a sequence of \( n \) numbers, and \( goal \) is either :maximize or :minimize. Returns an f64vector \( x \) of length \( n \), such that it maximizes or minimizes the inner product \( c \cdot x \), while satisfying the constraint \( Ax \leq b \) and \( x \geq 0 \).

The array can be any type of arrays (see Section 9.1 [Arrays], page 366) and the two sequences can be any type of sequence, as far as they only contain real numbers. Internally it uses f64vector and f64array for calculation. The arguments aren’t mutated during computation.

You can have negative entries in \( b \); the procedure uses two-phase simplex method in that case.

If you have a ‘\( >\)’ constraint, you can negate both sides to make it a ‘\( <=\)’ constraint. If you have an equality in the constraint, you can either use it to reduce one variable, or add both ‘\( >\)’ and ‘\( <=\)’ constraints.

If there can be multiple solutions, the procedure returns one of them. If there’s no solutions, it returns #f.

```scheme
;; Maximize x1 + x2 under constraints of
;;  x1 + 0.5*x2 <= 2.0 and
;;  3*x1 + 2*x2 <= 12.0

(simplex-solve '#,(<array> (0 2 0 2) 1 0.5 3 2)
  '#f64(2.0 12.0)
  :maximize '#(1 1))
```
12.39 os.windows - Windows support

This module is only available on Windows-native Gauche, and provides Windows-specific procedures. You can check `gauche.os.windows` feature with `cond-expand` macro (see Section 4.12 [Feature conditional], page 75) to conditionalize windows-specific code.

```
(cond-expand
  [gauche.os.windows (use os.windows)
   ... Windows-specific code ...]
[else
   ... Unix code ...])
```

Currently there aren’t enough procedures provided here, but eventually we want to support simple scripting on Windows.

Unless otherwise noted, when Windows API returns an error value, a `<system-error>` condition is thrown.

12.39.1 Windows dialogs

Currently we only have MessageBox API.

```
sys-message-box window message :optional caption flags
{os.windows} Calls Windows MessageBox API. The `window` argument should be a handle for a window, or `#f`; at the moment we don’t provide any API that retrieves window handles, so you should always pass `#f` here. The `message` argument takes a string for the content of the message box. Optional `caption` argument takes a string to be used in the window title.

The `flags` argument is an integer; it should be `logior` of values from one or more of the following groups. See the Windows reference manual for the details.

- **Buttons**
  - `MB_ABORTRETRYIGNORE`, `MB_CANCELCLOSECANCEL`, `MB_HELP`, `MB_OK` (default), `MB_OKCANCEL`, `MB_RETRYCANCEL`, `MB_YESNO`, `MB_YENOCANCEL`

- **Icon**
  - Default is no icon. Possible values: `MB_ICONEXCLAMATION`, `MB_ICONWARNING`, `MB_ICONINFORMATION`, `MB_ICONASTERISK`, `MB_ICONQUESTION`, `MB_ICONSTOP`, `MB_ICONERROR`, `MB_ICONHAND`

- **Default button**
  - `MB_DEFBUTTON1` (default), `MB_DEFBUTTON2`, `MB_DEFBUTTON3`, `MB_DEFBUTTON4`

- **Modality**
  - `MB_APPLMODAL` (default), `MB_SYSTEMMODAL`, `MB_TASKMODAL`

- **Other options**
  - `MB_DEFAULTDESKTOPONLY`, `MB_RIGHT`, `MB_RTLREADING`, `MB_SETFOREGROUND`, `MB_TOPMOST`, `MB_SERVICE_NOTIFICATION`

Return value is one of the following integer constants, indicating which button is pressed:

- `IDABORT`, `IDCANCEL`, `IDCONTINUE`, `IDIGNORE`, `IDNO`, `IDOK`, `IDRETRY`, `IDTRYAGAIN`, or `IDYES`

12.39.2 Windows console API

Most of these procedures corresponds to Windows Console API one-to-one. See the Windows reference for the detail description of what each API does.
Attaching and detaching

sys-alloc-console  [Function]
sys-free-console  [Function]
    [Windows] {os.windows} Calls AllocConsole and FreeConsole, respectively.

sys-generate-console-ctrl-event event pgid  [Function]
    [Windows] {os.windows}

CTRL_C_EVENT  [Constant]
CTRL_BREAK_EVENT  [Constant]
    [Windows] {os.windows}

Console codepage

sys-get-console-cp  [Function]
sys-get-console-output-cp  [Function]
sys-set-console-cp codepage  [Function]
sys-set-console-output-cp codepage  [Function]
    [Windows] {os.windows}

sys-get-console-cursor-info handle  [Function]
sys-set-console-cursor-info handle size visible  [Function]
    [Windows] {os.windows}

sys-set-console-cursor-position handle x y  [Function]
    [Windows] {os.windows}

Console mode

sys-get-console-mode handle  [Function]
sys-set-console-mode handle mode  [Function]
    [Windows] {os.windows}

ENABLE_LINE_INPUT  [Constant]
ENABLE_ECHO_INPUT  [Constant]
ENABLE_PROCESSED_INPUT  [Constant]
ENABLE_WINDOW_INPUT  [Constant]
ENABLE_MOUSE_INPUT  [Constant]
ENABLE_PROCESSED_OUTPUT  [Constant]
ENABLE_WRAP_AT_EOL_OUTPUT  [Constant]
    [Windows] {os.windows}

Screen buffer

sys-create-console-screen-buffer desired-access share-mode inheritable  [Function]
    [Windows] {os.windows}

GENERIC_READ  [Constant]
GENERIC_WRITE  [Constant]
    [Windows] {os.windows}

FILE_SHARE_READ  [Constant]
FILE_SHARE_WRITE  [Constant]
    [Windows] {os.windows}
sys-set-console-active-screen-buffer \textit{handle} \quad \text{[Function]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

sys-scroll-console-screen-buffer \textit{handle \textit{scroll-rectangle}} \quad \text{[Function]} \\
\hspace{1em} \textit{clip-rectangle} \textit{x y fill} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

<\text{win:console-screen-buffer-info}> \quad \text{[Class]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

\begin{verbatim}
size.x           \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
size.y           \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
cursor-position.x \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
cursor-position.y \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
attributes       \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
window.left      \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
window.top       \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
window.right     \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
window.bottom    \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
maximum-window-size.x \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
maximum-window-size.y \quad \text{[Instance Variable of <win:console-screen-buffer-info>]} \\
\end{verbatim}

FOREGROUND_BLUE \quad \text{[Constant]} \\
FOREGROUND_GREEN \quad \text{[Constant]} \\
FOREGROUND_RED \quad \text{[Constant]} \\
FOREGROUND_INTENSITY \quad \text{[Constant]} \\
BACKGROUND_BLUE \quad \text{[Constant]} \\
BACKGROUND_GREEN \quad \text{[Constant]} \\
BACKGROUND_RED \quad \text{[Constant]} \\
BACKGROUND_INTENSITY \quad \text{[Constant]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

sys-get-console-screen-buffer-info \textit{handle} \quad \text{[Function]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

sys-get-largest-console-window-size \textit{handle} \quad \text{[Function]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

sys-set-screen-buffer-size \textit{handle \textit{x y}} \quad \text{[Function]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

\textbf{Console input/output}

<\text{win:input-record}> \quad \text{[Class]} \\
\hspace{1em} \text{[Windows] \{os.windows\}}

\begin{verbatim}
event-type       \quad \text{[Instance Variable of <win:input-record>]} \\
key.down         \quad \text{[Instance Variable of <win:input-record>]} \\
key.repeat-count \quad \text{[Instance Variable of <win:input-record>]} \\
key.virtual-key-code \quad \text{[Instance Variable of <win:input-record>]} \\
key.virtual-scan-code \quad \text{[Instance Variable of <win:input-record>]} \\
key.unicode-char \quad \text{[Instance Variable of <win:input-record>]} \\
key.ascii-char   \quad \text{[Instance Variable of <win:input-record>]} \\
\end{verbatim}
key.control-key-state
mouse.x
mouse.y
mouse.button-state
mouse.control-key-state
mouse.event-flags
window-buffer-size.x
window-buffer-size.y
menu.command-id
focus.set-focus

sys-get-number-of-console-input-events handle
[Windows] {os.windows} [Function]

sys-get-number-of-console-mouse-buttons
[Windows] {os.windows} [Function]

sys-peek-console-input handle
[Windows] {os.windows} [Function]

sys-read-console-input handle
[Windows] {os.windows} [Function]

sys-read-console handle buf
[Windows] {os.windows} [Function]

sys-read-console-output handle buf w h x y region
[Windows] {os.windows} [Function]

sys-read-console-output-attribute handle buf x y
[Windows] {os.windows} [Function]

sys-read-console-output-character handle len x y
[Windows] {os.windows} [Function]

sys-set-console-text-attribute handle attr
[Windows] {os.windows} [Function]

sys-set-console-window-info handle absolute window
[Windows] {os.windows} [Function]

sys-write-console handle string
[Windows] {os.windows} [Function]

sys-write-console-output-character handle string x y
[Windows] {os.windows} [Function]

sys-fill-console-output-character handle char len x y
[Windows] {os.windows} [Function]

sys-fill-console-output-attribute handle attr len x y
[Windows] {os.windows} [Function]

sys-flush-console-input-buffer handle
[Windows] {os.windows} [Function]

sys-get-console-title
[Windows] {os.windows} [Function]

sys-set-console-title string
[Windows] {os.windows} [Function]
Standard handles

sys-get-std-handle which [Function]
sys-set-std-handle which handle [Function]

[Windows] {os.windows}

STD_INPUT_HANDLE {Constant}
STD_OUTPUT_HANDLE {Constant}
STD_ERROR_HANDLE {Constant}

12.40 parser.peg - PEG parser combinators

parser.peg [Module]

This module implements a parser combinator library to build parsers based on Parsing Expression Grammar, or PEG.

PEG is a formal grammar to define a language, like regular expressions or context-free grammars. An interesting characteristic of PEG is that it can be directly mapped to a recursive decent parser, which is exactly what this library does—each production rule is a Scheme expression that takes parsers and returns a combined parser. One advantage of this approach is that you can freely mix ordinary Scheme code within the parser, that is, there’s no special “parser description language” distinct from the base Scheme language, nor you need to run separate tools like parser generators to obtain a runnable parser code.

Although PEG can directly parse the character string, the parser combinators are not tied to it. In fact, most of the combinators work transparently for any sequence of tokens, where the exact meanings of tokens depend on the application; you can have separate lexer that generates token sequence that PEG parser can parse, for example.

This library is specifically written to get a good performance on Gauche. The parser created by parser.peg is no slower than the parser written manually from scratch. However, you have to watch out some traps; see Section 12.40.10 [PEG performance tips], page 917, for the details.

12.40.1 Walkthrough

In this section we cover the basic concepts and tools of parser.peg. The code of examples is in examples/pegintro.scm if you have the source tree of Gauche.

In parser.peg, a parser is merely a Scheme procedure that takes a list of tokens as an argument and returns a result (well, in fact, it returns three values, but we’ll go into the details later.)

Typically you don’t need to write parser procedures directly. Instead, you can use procedures that generates parsers. A parser can be as simple as the following, which accepts a character \a.

($) . \a) ; ⇒ a parser

Here, accept means the parser checks if the head of input has a character \a, and if it is, it succeeds, and if not, it fails.

A parser can be invoked by a parser driver. For example, you can use peg-parse-string to invoke the above parser on a string:

gosh> (peg-parse-string ($) . \a) "abc")
\a

The parsing succeeds, and returns the matched value—\a in this case. If the parser can’t accept the input, the driver throws an error <parse-error>.

gosh> (peg-parse-string ($char \a) "xyz")
A parser can also be constructed by combining simpler parsers, using *parser combinators*. For example, $\$seq$ takes zero or more parsers and apply them sequentially, returning the last result.

```
gosh> (peg-parse-string ($seq ($. #\a) ($. #\b) ($. #\c)) "abc")
#\c
```

The combinator $\$many$ takes a parser and returns a new parser that accepts zero or more occurrence of the string the original parser accepts.

```
gosh> (peg-parse-string ($many ($. #\a)) "aaaaabc")
(#\a #\a #\a #\a #\a)

(gosh> (peg-parse-string ($many ($. #\a)) "xxxxxyz")
())
```

A parser is just an ordinary Scheme procedure, so it can be bound to a variable, then can be used to construct more complex parsers.

```
(define digits ($many1 ($. #\[d\])))
(define ws ($many_ ($. #\[s\])))
(define separator ($seq ws ($. #\, ws))
```

I leave explanation of $\$many1$ and $\$many_ for the later section, but you may be able to guess what those parsers do; $\text{digits}$ accepts a sequence of one or more digits, and $\text{ws}$ accepts sequence of zero or more whitespaces. The $\text{separator}$ parser accepts a comma, optionally surrounded by whitespaces.

The $\text{digits}$ parser returns a list of accepted characters:

```
gosh> (peg-parse-string digits "12345")
(#\1 #\2 #\3 #\4 #\5)
```

Can we create a parser that returns an integer as a parsed result? Yes, we can use the $\$let$ macro.

```
(define integer
($let ([ds digits])
($return (x->integer (list->string ds)))))
```

The $\$let$ works somewhat like \text{and-let*}; it takes a form of ($\$let ([var parser] ...) expr ...), applying the parsers in order, binding the result of each parser to var. If any of the parser fails, the entire parser created by $\$let$ macro fails. When all the parser succeeds, each result is bound to var and expr ... are evaluated. The last expr must yield a parser.

The $\$return$ procedure creates a parser that doesn’t consume input, always succeeds and returns the given value. The name is taken from Haskell’s monads. Note that is is just an ordinary procedure and not like a control-transfer syntax like traditional language’s return. You may think it just as type conversion procedure from a Scheme object to a parser.

```
gosh> (peg-parse-string integer "12345")
12345
```

Now you can combine those parsers to build more complex one, such as a comma-separated list of integers:

```
(define integers1 ($seq integer
($many ($seq separator integer)))))
```

```
gosh> (peg-parse-string integers1 "123, 456, 789")
(456 789)
```

Oops, where’s 123? Well, remember that $\$seq$ discards the results but the last one. We can use $\$let$ again to keep all the results.

```
(define integers2 ($let ([n integer]
```

```
(ns ($many ($seq separator integer)))

($return (cons n ns)))

(gosh> (peg-parse-string integers2 "123, 456, 789")
(123 456 789)

(Unlike let where the order of its init expressions are not defined, $let guarantees the
parsers are applied sequentially. The reason it is not called $let* is the scope; we also have
$let*, which we’ll explain shortly.)

Another way to gather the results of parsers is a combinator $lift. It is used as ($lift
proc parser ...), where proc is an ordinary procedure which receives the result of parser ... as arguments. The return value of proc becomes the result of the entire parser. Unlike $let, proc doesn’t need to return a parser.

(define integers3 ($lift cons
integer
($many ($seq separator integer))))

The parsers so far don’t handle the case when the list contains no integers. Using $or
combinator, which represents a choice, we can modify it to handle zero-element case.

(define integers4 ($or ($let ([n integer]
[ns ($many ($seq separator integer))])
($return (cons n ns)))
($return '()))))

By the way, “list of stuff separated by something” is a very common pattern, so we can
extract the pattern to name it:

(define (sep-by stuff separator)
($or ($let ([n stuff]
[ns ($many ($seq separator stuff))])
($return (cons n ns)))
($return '()))))

Then the list of integers can be written this simple:

(define integers5 (sep-by integer separator))

In fact, parser.peg provides $sep-by to do the above, but we’ve just shown the definition to demonstrate the power of the combinatorial approach; you can use ordinary procedural abstraction to factor out common patterns.

There’s one catch in the $or form. It tries the next alternative only when the parser fails
without consuming the input. Once the input is consumed, $or commits to that choice. For
example, the following fails even if the input seems to match the second alternative:

(define paren ($.
#\())
(define thesis ($.
#\)))

(peg-parse-string ($or ($seq paren ($."ab") thesis)
($seq paren ($."cd") thesis))
"(cd)"
⇒ *** PARSE-ERROR: expecting ab at 1, but got #\c

It’s because when $or tries the first branch, it reads the initial open paren from the input, so $or commits to the first branch. When the branch fails, $or doesn’t bother to try the second branch. (In other words, $or does not backtrack.)

You may factor out the common prefix:

($seq paren

But it may complicates the syntax, and it is not always trivial to factor out like above. The better way is to use the $try combinator: ($try p) runs a parser $p$, and if it fails, $try$ rolls back the input as if it didn’t consume input at all. Using with $or, you can do arbitrary lookahead and backtrack.

($or ($try ($seq paren ($"ab") thesis))
 ($seq paren ($"cd") thesis))

Now, let’s get back to the integer list example and make it more interesting. Suppose the list of integers are surrounded by parentheses, brackets or curly-braces. Opening one and closing one must correspond.

(define begin-list
 ($seq0 ($#[\(\[\{]) ws))

(define (end-list opener)
 ($seq ws (case opener
 [(#\() ($#\))]
 [(#\[) ($#\])]
 [(#\{) ($#\}))))

(define int-list
 ($let* ([opener begin-list] ;*1
 [ints ($sep-by integer separator)] ;*2
 [ (end-list opener) ] ;*3
 ($return ints)))

The opening bracket is parsed by the parser begin-list. The $seq0 combinator is similar to seq, but returns the result of the first parser instead of the last one (it’s begin0 to begin).

The closing bracket must match the opening one, so end-list is a procedure that takes the opening bracket and returns a suitable parser to accept the corresponding closing bracket.

The int-list first parses the opening bracket by begin-list and bind the result to opener (*1). Then goes to parse comma-separated integers (*2) and bind the result list to ints. Finally, it parses the matching closing bracket (*3). Note that the it omits the variable, since we don’t need the result of closing bracket parser.

Since we need to use the value of opener in the following bindings, we use $let* here, instead of $let. The difference is the scope of the parser expressions in the binding. Note that, however, $let is a lot more easier to optimize, so you want to use $let whenever possible.

Let’s see it works.

```
gosh> (peg-parse-string int-list "[123, 456, 789]")
(123 456 789)
gosh> (peg-parse-string int-list "{123, 456, 789}\}"
(123 456 789)
gosh> (peg-parse-string int-list "(123, 456, 789)"
*** PARSE-ERROR: expecting #\) at 14, but got #\}
```

The last example shows it rejects unmatched brackets.

What if we want a nested list? In BNF, we could write something like this:

```
list : begin-list (elem (separator elem)* )? end-list
elem : integer | list
```

The straight translation would be he following.

```
;; First try
```
(define nested-list
  ($let* ([opener begin-list]
          [ints ($sep-by elem separator)]
          [ (end-list opener) ])
  ($return ints)))
(define elem ($or integer nested-list))

Let’s load it... Oops.
*** ERROR: unbound variable: elem
Stack Trace:
----------------------------------------
0 elem
[unknown location]
1 (eval expr env)
at "../lib/gauche/interactive.scm":267

We need the parser elem to construct nested-list, but we need the parser nested-list to construct elem. In lazy languages like Haskell this doesn’t matter, but we Schemers are eager!
The solution is to delay the parser construction until it is actually used. The $lazy form does the job:

(define nested-list
  ($lazy
    ($let* ([opener begin-list]
            [ints ($sep-by elem separator)]
            [ (end-list opener) ])
  ($return ints))))
(define elem ($or integer nested-list))

gosh> (peg-parse-string nested-list "(123, [456, {}, 789], 987)")
(123 (456 {} 789) 987)

Ok, we’re almost done. Our code can parse nested list of integers, checking bracket matches. But if you give an erroneous input, the message is cryptic and not helpful:
gosh> (peg-parse-string nested-list "(123, [456, {}, 789), 987")
*** PARSE-ERROR: expecting one of (#[0-9] #\]) at 19
Stack Trace:
----------------------------------------
0 (eval expr env)
at "../lib/gauche/interactive.scm":267

We could check the reason of failure in the parser, and call $fail with more reasonable error message. Let’s replace end-list above with end-list2 below:

(define (end-list2 opener)
  (define expected
    (assv-ref '((#( . #\) (#\[ . #\]) (#\{ . #\})) opener))
  ($seq ws
    ($let ([closer ($. #\[")\]])
      (if (eqv? closer expected)
        ($return closer)
        ($fail (format "Mismatched closing bracket. ’"c’ expected, \ but got ’"c’ expected closer))))))

You also have to change nested-list and elem to use end-list2:

(define nested-list2
($lazy
  ($let* ([opener begin-list]
    [ints ($sep-by elem2 separator)]
    [ (end-list2 opener) ])
  ($return ints))))

(define elem2 ($or integer nested-list2))

(You could redefine end-list, but if you do so, don't forget to re-evaluate the definitions of nested-list and elem. It's because the combinators are calculated taking the value of other combinators when it's defined, unlike typical procedural approach where you redefine one procedure and other procedures will refer to the updated version after that.)

And now you see this error:

gosh> (peg-parse-string nested-list2 "(123, [456, {}, 789), 987")
*** PARSE-ERROR: Mismatched closing bracket. ']' expected, but got ')' at 20
Stack Trace:

---------
0 (eval expr env)
  at "/lib/gauche/interactive.scm":267

For further examples, you can take a look at some libraries in the Gauche source tree that use parser.peg:

- lib/rfc/json.scm
- lib/text/edn.scm
- lib/www/css.scm

12.40.2 Parser drivers

peg-run-parser parser list  [Function]
{parser.peg} Apply parser on the input list. If parser accepts it, returns two values—the result of parser, and the rest of the input. If parser fails, raise <parse-error>.

The result of parse is a value yielded by parser passed to rope-finalize, so any unfinalized rope is finalized before being returned. See Section 12.40.5 [PEG ropes], page 912, for the details.

Typically, you don’t want to have entire input as a list beforehand, so you pass a lazy sequence as list (see Section 6.18.2 [Lazy sequences], page 241). If the input is a string or a port, convenience procedures are defined.

Input doesn’t need to be a character list. You may, for example, have a separate lexer that generates a (lazy) list of tokens, and let parser parse them.

peg-parse-string parser string :optional cont  [Function]
peg-parse-port parser iport :optional cont  [Function]
{parser.peg} Convenience wrappers of peg-run-parser that takes input from string or iport.

Without cont argument or it is #f, it returns the parsed result and discards the rest of the input. (Hint: To make sure there’s no garbage following, use $eos parser.)

If you want to keep parsing after parser accepts its input, pass cont a procedure; it must take two arguments, the result of parser and a lazy sequence of the rest of the input. What cont returns will be the result of these procedures.

It is an error to pass other values to cont.
**Chapter 12: Library modules - Utilities**

**Function**

`peg-parser->generator parser list`

{`parser.peg`} This is useful when you need to apply `parser` repeatedly over the input list. Returns a generator that generates the parsed result one match at a time.

The same input can be accepted by by `(peg-run-parser ($many parser) list)`, but this one won’t return until all input is consumed. On the other hand,

**Condition type**

`<parse-error>`

{`parser.peg`} An condition type raised by the parser driver when the given parser failed ultimately. Inherits `<error>`. (Note that each parser won’t throw this; one parser’s failure doesn’t necessarily mean the entire parser fails. It’s the driver that recognizes the ultimate failure and raise this condition.)

The following slots are available:

**message**

[Instance Variable of `<parse-error>`]

This slot is inherited from `<error>`. Contains string error message.

**position**

[Instance Variable of `<parse-error>`]

The position the failure occurred.

The exact meaning of this value depends on how you call the parser driver.

At minimum, when you call it on a bare list or a string/port (using `peg-parse-string/peg-parse-port`), the parser driver counts the number of elements took from the input and put it here, for it is the only available information. Note that the number may not be what you want, e.g. if you start parsing in the middle of the stream.

If you’re parsing from a sequence with positions, this slot contains an instance of `<sequence-position>`, which may have line & column numbers or a source file name. See Section 9.15.3 [Lazy sequence with positions], page 454, for the details.

The user code should expect both cases.

**type**

[Instance Variable of `<parse-error>`]

The type of failure. It’s either one of the followng symbols:

**fail-expect**

The parser expects one of the objects in `objects` slot, but got a different one (stored in `token`).

**fail-unexpect**

The parser isn’t expecting any of the objects in `objects` slot, but got the one in `token`.

**fail-compound**

The parser failed multiple options. The `objects` slot contains a list of of (type . msg) where type is one of the `<parse-error>` types, and msg is the message associated with it.

**fail-message**

Other miscellaneous failure. The `objects` slot contains a mmmessage.

**fail-error**

Non-recoverable failure. Once this failure is generated, parser combinators don’t backtrack and let the entire parsing fail. This type of failure can be generated by $rause parser constructor and $cut combinator. The `objects` slot contains a cons whose car is a symbol (error tag) and whose cdr is a list of (type . msg). The cdr part is the same as fail-compound.

**objects**

[Instance Variable of `<parse-error>`]

Value of this slot depends on the value of type slot.
rest [Instance Variable of <parse-error>]
The remaining input token stream, beginning from the point when the failure occur.

token [Instance Variable of <parse-error>]
The token at the head of input when failure occur. If input already reached at the end,
this slot is set to #<eof>.
This is the same as the car of rest slot when there’s more input.

12.40.3 What is a PEG parser, really?
A PEG parser is a Scheme procedure that takes a list of items as an input, and returns three
values:
• Failure type. If the parser successfully accepts the input, this value is #f. If the parser fails
to accept the input, this value is either one of the symbols fail-expect, fail-unexpect,
fail-compound and fail-message. It corresponds to the type slot of <parse-error>, and
determines the meaning of the second return value.
• Value. If the parser successfully accepts the input, this value is the parser’s “result”,
sometimes called a semantic value. If the parser fails to accept the input, this value contains
the hint of the failure. It corresponds to the objects slot of <parse-error>; see the
documentation of <parse-error> for the details (see Section 12.40.2 [PEG parser drivers],
page 907).
• The rest of the input list.

Typically you don’t need to write a parser as a procedure; instead, you can use one of the
parser builders and combinators described in the following sections.
We do provide a few utilities to write a parser from scratch, in case you need to do so.

return-result value rest [Function]
{parser.peg} Call this at the tail position of the parser when it succeeds, with value
for the semantic value and rest for the rest of input. This is the same as (values #f value rest),
but clearer to show the intention.

return-failure(expect) objs rest [Function]
return-failure/unexpect objs rest [Function]
return-failure/message msg rest [Function]
return-failure/compound fails rest [Function]
{parser.peg} Call one of these at the tail position of the parser when it fails. The first
argument will be in objects slot of <parse-error>. The second argument should be a list
of input, with the first element being a token that can’t be accepted.

• If you’re expecting some types of objects (objs) but got anything else, call
return-failure/expect. It generates fail-expect type failure.
• If you’re not expecting some types of objects (objs) but got one, call
return-failure/unexpect. It generates fail-unexpect type failure.
• If you’re failing because of all of multiple choices fail, you can gather those failures into an
assoc list ((fail-type . objs) ...), and call return-failure/compound. It generates
fail-compound type failure.
• Finally, your failure can’t be categorized to one of the above, you can call
return-failure/message, with the message describing the reason of the failure.

return-failure type objs rest [Function]
{parser.peg} This should only be used to pass down the failure form other parser. See the
example in parse-success? entry below.
parse-success? r
  \{parser.peg\} Check the first return value of a parser to see if it is a success. A typical usage is to check another parser’s result and take actions accordingly:

  \(\text{receive (r v s) (parser input)}\)
  \(\text{(if (parse-success? r)}\)
  \(\text{(... do things after PARSER succeeds ...}\)
  \(\text{(return-result ...))}\)
  \(\text{(return-failure r v s))}\)

This is simply checking if \(r\) is \#f, but using this procedure indicates your intention clearly.

12.40.4 Primitive parser builders
Procedures and macros that create parsers.

\$\text{return \hspace{1em} val} \quad \{\text{parser.peg}\} \text{Returns a parser that always succeeds, without consuming input, and yields val as the result of parser.}\n
Frequently used in $\text{let}$ and $\text{let*}$’s body, but can be used anywhere a parser is expected.

\$\text{fail \hspace{1em} msg-string} \quad \{\text{parser.peg}\} \text{Returns a parser that always fails, without consuming input, and uses msg-string as the failure message.}\n
Frequently used to produce user-friendly error messages.

\$\text{raise \hspace{1em} msg-string} \quad \{\text{parser.peg}\} \text{Returns a parser that raises a non-recoverable failure with msg-string as the failure message.}\n
The difference from \$\text{fail}$ is that, if $\text{or}$ sees a failure created by \$\text{fail}$, it may try the remaining branches, while if it sees a failure created by \$\text{raise}$, no more branches are tried.

This can be used for better error reporting. If you detect the case that can’t be a valid input in deep in the parse tree, a normal failure would try other alternatives exhaustively and generate an error message itemizing all the failed possibilities. It is often difficult to see the real cause from such a message. With \$\text{raise}$, you can let the parser give up immediately.

See also \$cut combinator below, to convert a normal failure to non-recoverable failure.

\$\text{satisfy \hspace{1em} pred \hspace{1em} expect \hspace{1em} :optional result} \quad \{\text{parser.peg}\} \text{This corresponds to “semantic predicate” in PEG; a parser that can apply an arbitrary predicate on input.}\n
Returns a parser that works as follows:

- If the head of input stream satisfies \(pred\), call \(\text{(result head (pred head))}\) and yield its return value as the result of successful parsing. If result is omitted, it yields the head of input.
- Otherwise, the parsing fails with expect as the expected input.

If you just need a lookahead parser, you can use \$\text{assert}.$

\$. \hspace{1em} obj \quad \{\text{parser.peg}\} \text{Creates a parser that matches a Scheme object obj, which may be a character, a string, a char-set, or a symbol. If obj is a char-set, the parser matches any character in the set.}\n
The resulting parser is atomic, that is, it doesn’t consume input when it fails.
Chapter 12: Library modules - Utilities

$char c  
[Function]
$char-ci c
[Function]

{parser.peg} Returns a parser that accepts a single character, c. $char-ci ignores case. On success, the parser yields the input character. The resulting parser is atomic, that is, it doesn’t consume input when it fails.

$string str
[Function]
$string-ci str
[Function]
{parser.peg} Returns a parser that accepts an input that matches a string str. $string-ci ignores case. On success, the parser yields the matched string.

The parsing of string is atomic: When the parser fails, it doesn’t consume the input. That is, ($string "ab") is not the same as ($let ([a ($char #\a)] [b ($char #\b)]) ($return (string a b))).

$one-of cset
[Function]
$one-of obj-list
[Function]
{parser.peg} The first form returns a parser that accepts any character in the character set cset. In the second form, obj-list must be a list of either a character, a string, a character set or a symbol, and each one is matched with the same way as $.

On success, the parser yields the accepted object.

$none-of cset
{parser.peg} Returns a parser that accepts any character not in the character set cset. On success, the parser yields the accepted character.

$any
{parser.peg} Returns a parser that matches any one item, and yields the matched input item on success. It fails only when the input already reached at the end.

$eos
{parser.peg} Stands for “end of stream”. Returns a parser that matches the end of input. It never consumes input.

$match1 pattern result
$match1 pattern
$match1 pattern (=> fail) result
$match1* pattern result
$match1* pattern
$match1* pattern (=> fail) result

{parser.peg} The pattern matcher macro match-let1 lifted to the parser. See Section 12.87 [Pattern matching], page 1024, for the details of supported pattern.

The macro $match1 returns a parser that takes one item from the input stream, and see if it matches pattern. If it matches, evaluate result within an environment where pattern variables are bound to matched content, and the parser yields the value of result. If the input doesn’t match pattern, or the input is empty, the parser fails without consuming input.

In the third form => must be a literal identifier and fail must be an identifier. The identifier fail is bound to a procedure that takes one string argument in result. You can call fail at the tail position of result to make the match fail, with the passed argument as the message. If fail is called, no input will be consumed.

(NB: The match macro in util.match has a similar feature, but it binds fail to a continuation that abandons the current match clause and go to try the next pattern. In $match1, fail is simply a procedure, so you have to call it at the tail position to make it work.)
The macro $\texttt{match1*}$ is similar to $\texttt{match1}$, except the entire input is matched pattern. It is useful to look into several items in input, instead of just one. Note that if you give a pattern that consumes arbitrary length of input (e.g. ($\texttt{match1* (a ...)}$), it will consume entire input.

These macros especially come handy when you have a token stream generated by a separate lexer—each token can have some structure (instead of just a character) and you can take advantage of $\texttt{match}$.

### 12.40.5 Ropes

Often you want to construct a string out of the results of other parsers. It can be costly to construct strings eagerly, for a string may be just an intermediate one to be a part of a larger string. We provide a lightweight lazily string construction mechanism, called ropes.

A rope is either a character, a string or a pair of ropes. It allows O(1) concatenation. A rope becomes a string when finalized. The parser drivers such as $\texttt{peg-run-parser}$ automatically finalizes ropes in the parser result.

```
$\rightarrow\text{rope}$ parser ... [Function]
    {parser.peg} The parsers must yield either a character, a string, a rope, or $\texttt{#f}$ or (). This procedure returns a parser that matches parser ..., then gather the result into a rope. $\texttt{#f}$ and () in the results are ignored.

$\rightarrow\text{string}$ parser ... [Function]
    {parser.peg} This is a common idiom of ($\texttt{lift rope->string ($\rightarrow\text{rope}$ parser ...))}.

$\rightarrow\text{symbol}$ parser ... [Function]
    {parser.peg} Like $\rightarrow\text{string}$, but yields a symbol rather than a string.

$\text{rope->string}$ rope [Function]
    {parser.peg} Converts a rope to a string.

$\text{rope-finalize}$ obj [Function]
    {parser.peg} Converts any ropes in in obj into strings.

### 12.40.6 Choice, backtrack and assertion combinators

```
$\texttt{or}$ p1 p2 ... [Function]
$\texttt{or}$ p1 p2 ... $\texttt{:else}$ plast [Function]
    {parser.peg} Returns a choice parser. Tries the given parser in order on input. If one succeeds, immediately yields its result. If one fails, and does not consume input, then tries the next one. If one fails with consuming input, immediately fails. If p1 p2 ... don’t share the same prefix to match, you can let it fail as soon as one parser fails with consuming input. If more than one parsers do match the same prefix, you want to wrap them with $\texttt{try}$ except the last one.

If all of the parsers p1 p2 ... fail without consuming input, $\texttt{or}$ returns a compound failure of all the failures. You may wish to produce better error message than that. Putting $\texttt{fail}$ parser at the last doesn’t cut it, for $\texttt{fail}$ doesn’t consume input so all the previous failures would be compound. In such cases, you can use the second form—if the argument before the last parser is a keyword $\texttt{:else}$, then $\texttt{or}$ discards the previous failures.

(peg-parse-string ($\texttt{or}$ ($.$ "ab")
    ($.$ "cd")
    $\texttt{:else}$ ($\texttt{fail}$ "we want 'ab' or 'cd'")
    "ef")
⇒ PARSE-ERROR: 'ab' or 'cd' required at 0
$\textbf{try } p \{\text{parser.peg}\}$ Returns a parser that accepts the same input the parser $p$ accepts, but when $p$ fails the returned parser doesn’t consume input. Used with $\textbf{or}$, you can explicitly implement a backtrack behavior.

$\textbf{optional } p : \text{optional fallback} \{\text{parser.peg}\}$ Returns a parser that tries $p$ on the input. If it succeeds, yielding its result. If it fails, it still succeeds, yielding $\text{fallback}$ as the result.

This is atomic; if $p$ fails, it doesn’t consume input.

$\textbf{assert } p \{\text{parser.peg}\}$ Returns a parser that accepts the same input as $p$ and returns its result on success, but never consumes the input. It can be used as a lookahead assertion.

$\textbf{not } p \{\text{parser.peg}\}$ Returns a parser that succeeds when $p$ fails, and that fails when $p$ succeeds. When $p$ succeeds, it yields an “unexpected” error. It never consumes input in either way. It can be used as a negative lookahead assertion.

$\textbf{expect } p \text{ msg-string} \{\text{parser.peg}\}$ Returns a parser that calls a parser $p$, and if it succeeds yields its result. If $p$ fails, fails with an error message that says expecting $\text{msg-string}$. Useful to produce user-friendly error messages.

$\textbf{cut } p \{\text{parser.peg}\}$ If $p$ fails, make the failure non-recoverable. It prevents the upstream $\textbf{or}$ and $\textbf{try}$ from backtracking and trying other choices, and makes the entire parsing fail immediately.

See also $\textbf{raise}$ above.

12.40.7 Sequencing combinators

$\textbf{seq } p1 p2 \ldots \{\text{parser.peg}\}$ Returns a parser that matches $p1$, $p2$, $\ldots$ sequentially. When all the parser succeeds, $\textbf{seq}$ returns the last result, while $\textbf{seq0}$ returns the first result. Fails immediately when one of the parsers fails.

$\textbf{between } p1 p2 p3 \{\text{parser.peg}\}$ Returns a parser that matches $p1$, $p2$ and $p3$ sequentially, and returns the result of $p2$.

$\textbf{list } p \ldots \{\text{parser.peg}\}$ Returns a parser that matches $p \ldots$, and returns the list of the results. $\textbf{list}$ uses the last parser’s result as the last cdr.

They are the same as ($\text{lift list } p \ldots$) and ($\text{lift list}^* p \ldots$), but we encounter this pattern frequent enough to have these.

$\textbf{bind } p f \{\text{parser.peg}\}$ The basic block of parser combinators; $p$ argument is a parser, and $f$ is a procedure that takes a Scheme value and returns a parser.

Returns a parser that first applies $p$ on the input, and if it succeeds, calls $f$ with the result of $p$, and applies the returned parser on the subsequent input.
This combinator, along with $\text{return}$ and $\text{fail}$, composes a MonadFail interface as in Haskell. Theoretically any combinators can be built on top of these three. In practice, however, it is not always easy to build things directly on top of $\text{bind}$, and more high-level forms such as $\text{let}$, $\text{let*}$ and $\text{lift}$ are frequently used.

**$\text{let}$ (binding ...) body ...** [Macro]

**$\text{let*}$ (binding ...) body ...** [Macro]

{parser.peg} Monadic binding form. Each binding can be one of the following forms:

- **(var parser)**
  
  Run the parser, and if it succeeds, bind its result to a variable var. If it fails, the entire $\text{let}$ or $\text{let*}$ immediately fails.

- **(parser)**
  
  The variable is omitted. The parser is run, and if it succeeds, its result is discarded and the next binding or body is evaluated. If it fails, the entire $\text{let}$ or $\text{let*}$ immediately fails.

- **parser**
  
  Same as above. This form can only be used if parser is just a variable reference.

Once all the parsers in binding ... succeeds, body ... are evaluated in the environment where var in bindings are bound to the parser results. The last expression of body must return a parser.

Unlike $\text{let}$, the parsers in binding ... are always applied to the input sequentially. The difference of $\text{let}$ and $\text{let*}$ is the scope. With $\text{let*}$, the variables bound in earlier binding can be used to construct the parser later.

This means $\text{let}$ can evaluate all the parsers beforehand, while $\text{let*}$ may need to construct parsers at the time of processing input. Creating a parser involves closure allocations, so you want to use $\text{let}$ whenever possible.

Note: $\text{let*}$ is similar to Haskell’s $\text{do}$ construct. We chose the name $\text{let}$ and $\text{let*}$, for it is easier to see it’s a binding form, and also Scheme already uses $\text{do}$ for loop construct.

**$\text{lift}$ f p ...** [Function]

**$\text{lift*}$ f p ...** [Function]

{parser.peg} Lifts a procedure f onto the parsers’ world.

In a pseudo type declaration, lift’s type can be understood as follows:

\[
\text{lift} :: (a \ b \ ... \rightarrow z) \ (\text{Parser} \ a) \ (\text{Parser} \ b) \ ... \rightarrow (\text{Parser} \ z)
\]

That is, \text{lift} creates a parser such that it first applies parsers on the input, and if all of them succeed, it calls f with the parsers’ results as arguments, and the return value of f becomes the whole parser’s result.

In other words, the following equivalence holds:

\[
(\text{lift} \ f \ p0 \ p1 \ ...)
\equiv (\text{let} \ ([r0 \ p0] \ [r1 \ p1] \ ...) \ (\text{return} \ (f \ r0 \ r1 \ ...)))
\]

It is sometimes simpler to use $\text{lift}$ instead of $\text{let}$. For example, the following code creates a parser that matches input with \(p0 \ p1 \ ...\) sequentially, then yields the list of the parser results:

\[
(\text{lift} \ \text{list} \ p0 \ p1 \ ...)
\]

Note that after all the parsers succeed, the whole parser is destined to succeed—the procedure f can’t make the parser fail. If you need to fail after all the parsers succeeds, use $\text{let}$ or $\text{let*}$.

**$\text{binding}$ parser-bind-form ... [(=\rightarrow \text{fail})] expr** [Macro]

**$\text{binding}$ parser-bind-form ... [(=\rightarrow \text{fail})] expr** [Macro]

{parser.peg} Each parser-bind-form may be a parser-yielding expression, except that you can insert a form ($\text{var}$ parser-expression) anywherer in it, where parser-expression is
an expression that yields a parser. The $: form is equivalent to just the parser-expression, except that its semantic value is bound to a variable var.

Each parser created by parser-bind-form is applied to the input in sequence. One of parser-bind-form fails, the entire parser immediately fails. If all of parser-bind-form succeeds, expr is evaluated in the environment where all the vars are bound to the corresponding parser expression.

Since $binding walks entire parser-bind-form to look for $: forms, you can’t have nested $binding form inside parser-bind-form.

If the parser expression associated with var fails, or never executed, the var is bound to #<undef>. If the parser expression succeeds multiple times, var holds the last value. Also, var can appear more than one places; it holds the last bound value.

The value of expr form becomes the semantic value of the entire parser.

$lbinding is a shorthand of ($lazy ($binding ...)).

The optional (=> fail) form before expr is similar to the one with $match. If given, fail, which must be an identifier, is bound to a procedure that returns failure. You should call it in the tail position of expr to indicate failure. It can be

- (fail message) : Returns fail-message type failure, with a string message as the message.
- (fail tag message) : Returns fail-error type (non-recoverable) failure, where tag must be a symbol error (in future, different tags will be supported).

$fold-parsers proc seed ps
{parser.peg} Ps is a list of parsers. Apply those parsers sequentially on the input, passing around the seed value. That is, if we let v0, v1 . . . vn be the result of each parsers in ps, it returns (proc vn (... (proc v2 (proc v1 seed))...)).

If any of the parser in ps fails, $fold-parsers fails at that point.

Conceptually, it can be written as follows:

(define ($fold-parsers proc seed ps)
  (if (null? ps)
      ($return seed)
      ($let ((v (car ps)))
        ($fold-parsers proc (proc v seed) (cdr ps)))))

But we use more efficient implementation.

$fold-parsers-right proc seed ps
{parser.peg} Similar to $fold-parsers, but the folding by proc right to left. That is, if we let v0, v1 . . . vn be the result of each parsers in ps, it returns (proc v1 (proc v2 (... (proc vn seed))...)).

If any of the parser in ps fails, $fold-parsers-right fails at that point.

12.40.8 Repetition combinators

$many p :optional min max
$many_ p :optional min max
{parser.peg} Without optional arguments, returns a parser that accepts zero or more repetition of p. On success, $many yields a list of matched results, while $many_ doesn’t keep the results (and faster).

Optional min and max must be nonnegative integers and limit the number of occurrences of p. The numbers are inclusive. For example, ($many ($) #\a) 3) accepts three or more #\a’s, and ($many ($) #\a) 2 4) accepts aa, aaa and aaaa.
Note that $\texttt{many}$ may fail if the input partially matches $p$.

\[
(\text{peg-parse-string} \ (\text{many} \ (\text{seq} \ (\. \ #\a) \ (\. \ #\b))) \ "ababcd") \\
\Rightarrow \ (\#\b \ #\b)
\]

\[
(\text{peg-parse-string} \ (\text{many} \ (\text{seq} \ (\. \ #\a) \ (\. \ #\b))) \ "ababac") \\
\Rightarrow \ \text{*** PARSE-ERROR: expecting #\b at 5, but got #\c}
\]

If you want to stop $\texttt{many}$ at the first two occurrences in the latter case, use $\texttt{try}$:

\[
(\text{peg-parse-string} \ (\text{many} \ (\text{try} \ (\text{seq} \ (\. \ #\a) \ (\. \ #\b)))) \ "ababac") \\
\Rightarrow \ (\#\b \ #\b)
\]

$\texttt{many1}$

$\texttt{many1 p :optional max}$

\{parser.peg\} Returns a parser that accepts one or more occurrences of $p$. On success, $\texttt{many1}$ yields a list of results of $p$, while $\texttt{many}$ discards the results of $p$ and faster. If $\texttt{max}$ is given, it specifies the maximum number of matches.

Same as ($\texttt{many p 1 max}$) and ($\texttt{many p 1 max}$). Provided as a common pattern.

$\texttt{repeat}$

$\texttt{repeat p n}$

$\texttt{repeat p n}$

\{parser.peg\} Returns a parser that accepts exactly $n$ occurrences of $p$. On success, $\texttt{repeat}$ yields a list of results of $p$, while $\texttt{repeat}$ discards the results of $p$ and faster.

Same as ($\texttt{many p n n}$) and ($\texttt{many p n n}$). Provided as a common pattern.

$\texttt{many-till}$

$\texttt{many-till p pe :optional min max}$

$\texttt{many-till p pe :optional min max}$

\{parser.peg\} Returns a parser that accepts repetition of $p$, until it sees input that accepts $pe$. On success, $\texttt{many-till}$ yields a list of results of $p$, while $\texttt{many-till}$ discards the results of $p$ and faster.

(\text{define comment (\text{seq} (\. ";") (\text{many-till} (\text{any}) (\. "\n"))))

$\texttt{sep-by}$

$\texttt{sep-by p psep :optional min max}$

$\texttt{end-by p psep :optional min max}$

$\texttt{sep-end-by p psep :optional min max}$

\{parser.peg\} These combinators match repetition of $p$ separated by $psep$, such as comma-separated values. Returns the list of results of $p$. Optional $\texttt{min}$ and $\texttt{max}$ are integers that limits the number of repetitions.

These three differ only on treatment of the last separator; $\texttt{sep-by}$ accepts strictly infix syntax, that is, the input must not end with the separator; while $\texttt{end-by}$ accepts strictly suffix syntax, that is, the input must end with the separator; $\texttt{sep-end-by}$ makes the last separator optional.

$\texttt{chain-left}$

$\texttt{chain-left p op}$

$\texttt{chain-right p op}$

\{parser.peg\} Returns a parser that parsers left-assosiative and right-associative operators, respectively.

The term of expression is parsed by a parser $p$, and the operator is parsed by $op$.

12.40.9 Miscellaneous combinators

$\texttt{parameterize}$

$\texttt{parameterize ((param expr \ldots) parser \ldots}$

\{parser.peg\} Returns a parser that runs $parser \ldots$, while altering the parameter values of $param \ldots$ with the result of $expr \ldots$, like $\texttt{parameterize}$. The $parser \ldots$ are run as if in $\texttt{seq}$, so only the value of them is returned on success.
You can’t use ordinary `parameterize`, since such parameterization takes effect on parser construction time, and not when the parser parsing the input.

```scheme
$debug name p
  {parser.peg} Parses the same input as p, but reports when it is parsing the input, and the result, just like `debug-print`.
You can’t use `debug-print` directly, for it will take effect on the parser construction time, not when the input is parsed.
```

```scheme
$lazy p
  {parser.peg} Returns a parser that works the same as p, but delays evaluation of p until needed. It is useful when you define mutually recursive parsers.
```

### 12.40.10 Performance

#### 12.41 rfc.822 - RFC822 message parsing

**rfc.822**

Defines a set of functions that parses and constructs the “Internet Message Format”, a text format used to exchange e-mails. The most recent specification can be found in RFC5322. The format was originally defined in RFC 822, and people still call it “RFC822 format”, hence I named this module. In the following document, I also refer to the format as “RFC822 format”.

**Parsing message headers**

**rfc822-read-headers iport :key strict? reader**

```scheme
{rfc.822} Reads RFC822 format message from an input port `iport`, until it reaches the end of the message header. The header fields are broken into a list of the following format:

```
((name body) ...)
```

- `Name . . .` are the field names, and `body . . .` are the corresponding field body, both as strings. Field names are converted to lower-case characters. Field bodies are not modified, except the folded line is unfolded. The order of fields are preserved.

- By default, the parser works permissively. If EOF is encountered during parsing header, it is taken as the end of the message. And if a line that doesn’t consist neither continuing (folded) line nor start a new header field, it is simply ignored. You can change this behavior by giving true value to the keyword argument `strict?`; then the parser raises an error for such a malformed header.

- The keyword argument `reader` takes a procedure that reads a line from `iport`. Its default is `read-line`, which should be enough for most cases.

**rfc822-header->list iport :key strict? reader**

```scheme
{rfc.822} This is an old name of `rfc822-read-headers`. This is kept for the backward compatibility. The new code should use `rfc822-read-headers` instead.
```

**rfc822-header-ref header-list field-name :optional default**

```scheme
{rfc.822} An utility procedure to get a specific field from the parsed header list, which is returned by `rfc822-read-headers`.

- `Field-name` specifies the field name in a lowercase string. If the field with given name is in `header-list`, the procedure returns its value in a string. Otherwise, if `default` is given, it is returned, and if not, `#f` is returned.

- This procedure can actually be used not only for the result of `rfc822-read-headers`, but for retrieving a value keyed by strings in a list-of-list structure: `((name value option ...) ...)`.  
```
For example, the return value of parse-cookie-string can be passed to rfc-822-header-ref (see Section 12.43 [HTTP cookie handling], page 924, for parse-cookie-string).

\[
\begin{align*}
\text{rfc822-header-ref} & \ (("from" \ "foo@example.com") \ ("to" \ "bar@example.com")) \ \\
& \ "from") \\
\Rightarrow & \ "foo@example.com"
\end{align*}
\]

;; If no entry matches, #f is returned by default
\[
\begin{align*}
\text{rfc822-header-ref} & \ (("from" \ "foo@example.com") \ ("to" \ "bar@example.com")) \ \\
& \ "reply-to") \\
\Rightarrow & \ #f
\end{align*}
\]

;; You can give the default value for no-match case
\[
\begin{align*}
\text{rfc822-header-ref} & \ (("from" \ "foo@example.com") \ ("to" \ "bar@example.com")) \ \\
& \ "reply-to" \ 'none) \\
\Rightarrow & \ none
\end{align*}
\]

;; By giving the default value, you can distinguish
;; the no-match case and there's actually an entry with value #f.
\[
\begin{align*}
\text{rfc822-header-ref} & \ (("from" \ "foo@example.com") \ ("reply-to" \ #f)) \\
& \ "reply-to" \ 'none) \\
\Rightarrow & \ #f
\end{align*}
\]

Basic field parsers

Several procedures are provided to parse "structured" header fields of RFC2822 messages. These procedures deal with the body of a header field, i.e. if the header field is "To: Wandering Schemer <schemer@example.com>", they parse "Wandering Schemer <schemer@example.com>".

Most of procedures take an input port. Usually you first parse the entire header fields by rfc822-read-headers, obtain the body of the header by rfc822-header-ref, then open an input string port for the body and use those procedures to parse them.

The reason for this complexity is because you need different tokenization schemes depending on the type of the field. Rfc2822 also allows comments to appear between tokens for most cases, so a simple-minded regexp won’t do the job, since rfc2822 comment can be nested and can’t be represented by regular grammar. So, this layer of procedures are designed flexible enough to handle various syntaxes. For the standard header types, high-level parsers are also provided; see "specific field parsers" below.

rfc822-next-token iport \ optional \ tokenizer-specs \ Function
\{RFC.822\} A basic tokenizer. First it skips whitespaces and/or comments (CFWS) from iport, if any. Then reads one token according to tokenizer-specs. If iport reaches EOF before any token is read, EOF is returned.

Tokenizer-specs is a list of tokenizer spec, which is either a char-set or a cons of a char-set and a procedure.

After skipping CFWS, the procedure peeks a character at the head of iport, and checks it against the char-sets in tokenizer-specs one by one. If a char-set that contains the character belongs to is found, then a token is retrieved as follows: If the tokenizer spec is just a char-
set, a sequence of characters that belong to the char-set consists a token. If it is a cons, the procedure is called with iport to read a token.

If the head character doesn’t match any char-sets, the character is taken from iport and returned.

The default tokenizer-specs is as follows:

\[
(\text{list}\ ((\text{cons}\ #\['\]\ rfc822-quoted-string))
\]
\[
(\text{cons}\ \ast\text{rfc822-atext-chars}\ast\ rfc822-dot-atom))
\]

Where rfc822-quoted-string and rfc822-dot-atom are tokenizer procedures described below, and *rfc822-atext-chars* is bound to a char-set of atext specified in rfc2822. This means rfc822-next-token retrieves a token either quoted-string or dot-atom specified in rfc2822 by default.

Using tokenizer-specs, you can customize how the header field is parsed. For example, if you want to retrieve a token that is either (1) a word constructed by alphabetic characters, or (2) a quoted string, then you can call rfc822-next-token by this:

\[
(\text{rfc822-next-token\ iport}'(#[[:alpha:]]\ #\['\]\ .\ rfc822-quoted-string)))
\]

Specific field parsers

rfc822-parse-date string

{rfc.822} Takes RFC-822 type date string, and returns eight values: year, month, day-of-month, hour, minutes, seconds, timezone, day-of-week.

Timezone is an offset from UT in minutes. Day-of-week is a day from sunday, and may be #f if that information is not available. Month is an integer between 1 and 12, inclusive. If the string is not parsable, all the elements are #f.

rfc822-date->date string

{rfc.822} Parses RFC822 type date format and returns SRFI-19 <date> object (see Section 11.7.4 [SRFI-19 Date], page 703). If string can’t be parsed, returns #f instead.

To construct rfc822 date string from SRFI-19 date, you can use date->rfc822-date below.
Message constructors

rfc822-write-headers  headers :key output continue check  [Function]
{rfc.822} This is a sort of inverse function of rfc822-read-headers. It receives a list of
header data, in which each header data consists of (<name> <body>), and writes them out in
RFC822 header field format to the output port specified by the output keyword argument.
The default output is the current output port.

By default, the procedure assumes headers contains all the header fields, and adds an empty
line in the end of output to indicate the end of the header. You can pass a true value to the
continue keyword argument to prevent this, enabling more headers can be added later.

I said “a sort of” above. That’s because this function doesn’t (and can’t) do the exact inverse.
Specifically, the caller is responsible for line folding and make sure each header line doesn’t
exceed the “hard limit” defined by RFC2822 (998 octets). This procedure cannot do the line
folding on behalf of the caller, because the places where line folding is possible depend on the
semantics of each header field.

It is also the caller’s responsibility to make sure header field bodies don’t have any char-
acters except non-NUL US-ASCII characters. If you want to include characters outside of
that range, you should convert them in the way allowed by the protocol, e.g. MIME. The
rfc.mime module (see Section 12.51 [MIME message handling], page 939) provides a conve-
nience procedure mime-encode-text for such purpose. Again, this procedure cannot do the
encoding automatically, since the way the field should be encoded depends on header fields.

What this procedure can do is to check and report such violations. By default, it runs several
checks and signals an error if it finds any violations of RFC2822. You can control this checking
behavior by the check keyword argument. It can take one of the following values:

:error  Default. Signals an error if a violation is found.
#f, :ignore  Doesn’t perform any check. Trust the caller.

procedure
When rfc822-write-headers finds a violation, the procedure is called with
three arguments; the header field name, the header field body, and the type of
violation explained below. The procedure may correct the problem and return
two values, the corrected header field name and body. The returned values
are checked again. If the procedure returns the header field name and body
unchanged, an error is signaled in the same way as :error is specified.

The third argument passed to the procedure given to the check argument is one of the
following symbols. New symbols may be added in future versions for more checks.

incomplete-string
Incomplete string is passed.

bad-character
Header field contains characters outside of US-ASCII or NUL.

line-too-long
Line length exceeds 998 octet limit.

stray-crlf
The string contains CR and/or LF character that doesn’t consist of proper line
folding.
Chapter 12: Library modules - Utilities

**date->rfc822-date**

{rfc.822} Takes SRFI-19 `<date>` object (see Section 11.7.4 [SRFI-19 Date], page 703) and returns a string of its rfc822 date representation. This is a reverse operation of `rfc822-date->date`.

### 12.42 rfc.base64 - Base64 encoding/decoding

**rfc.base64**

This module defines a few functions to encode/decode Base64 format, defined in RFC 4648 ([https://www.ietf.org/rfc/rfc4648.txt](https://www.ietf.org/rfc/rfc4648.txt)). It also supports Base32 and Base16 formats defined in it as well.

The common naming convention is as follows.

- **encoding-encode**, **encoding-decode**
  
  Port-based encoder/decoder. It reads input from the current input port, and emits encoded/decoded output to the current output port.

- **encoding-encode-message**
  
  Takes the input as either a string or a u8vector, and returns the encoded result as a string. Note that BaseN encoding always results readable strings.

- **encoding-decode-string-to**
  
  Takes the encoded string, and returns the result in either a u8vector or a string, as specified by the `target` argument. Note that the decoded result is a byte sequence in general, so a string may be incomplete.

#### Base64

**base64-encode**

{rfc.base64} Reads byte stream from the current input port, encodes it in Base64 format and writes the result character stream to the current output port. The conversion ends when it reads EOF from the current input port.

Newline characters can be inserted to keep the maximum line width to the value given to the `line-width` keyword argument. The default value of `line-width` is 76, as specified in RFC2045. You can give `#f` or zero to `line-width` to suppress line splitting.

Base64 can have different mappings in the digit of 62 and 63; the “standard” encoding uses `+` and `/`, and the “url-safe” encoding uses `-` and `_`. You can specify which to use by either `url-safe` or `digits`.

If a true value is given to `url-safe`, the input bytes will be encoded with “url-safe” encoding. Otherwise, you can give a two-letter string to `digits` argument, in which case the first character is used for digit 62 and the second character is used for digit 63. The `digits` argument is added to support SRFI-207 (see Section 11.47 [String-notated bytevectors], page 776).

If a true value is given to `omit-padding`, the trailing padding characters (`\#\=`) are omitted. (See RFC4648 Section 3.2 ([https://www.rfc-editor.org/rfc/rfc4648#section-3.2](https://www.rfc-editor.org/rfc/rfc4648#section-3.2))). The default is not omitting padding characters.

**base64-encode-message**

{rfc.base64} Encodes `message`, which must be either a string or a u8vector, with Base64 encoding and returns the result as a string. The meaning of keyword arguments are the same as `base64-encode` above.

```scheme
(base64-encode-message "Hau‘oli makahiki hou")
⇒ "SGF1J29saSBtYWthaGlraSBob3U=""
For the portable code, you may want to use `bytestring->base64` in `srfi.207` (see Section 11.47 [String-notated bytevectors], page 776).

**base64-decode** :key digits url-safe strict  
{rfc.base64} Reads character stream from the current input port, decodes it from Base64 format and writes the result byte stream to the current output port. The conversion ends when it reads EOF or the termination character (*).

The characters which do not in legal Base64 encoded character set are silently ignored by default. If a true value is given to the `strict` keyword argument, however, non-whitespace out-of-set characters in the input raises an error.

The input can lack the trailing padding character by default. However, if `strict` argument is true, an error is thrown unless the input has a proper number of padding characters.

The `digits` and `url-safe` keyword arguments can be used to choose different mappings of digit 62 and 63. See `base64-encode` above for the details.

**base64-decode-string-to** target string :key line-width digits url-safe  
omit-padding  
{rfc.base64} The `target` argument must be a class object `<string>` or `<u8vector>`. This procedure takes Base64-encoded input `string`, and returns the decoded result as an instance of `target`.

The meaning of keyword arguments are the same as `base64-decode` above.

(base64-decode-string-to <string> "SGF1J29saSBtYWthaGlraSBob3U=")  
⇒ "Hau'oli makahiki hou"
(base64-decode-string-to <u8vector> "SGF1J29saSBtYWthaGlraSBob3U=")  
⇒ #u8(72 97 117 39 111 108 105 32 109 97 107 97 104 105 107 105 32 104 111 117)

Note that if you ask a string to be returned, the result may be an incomplete string.

(base64-decode-string-to <string> "SPHy8w==")  
⇒ "#**"xf0;xf1;\xf2;\xf3;"

For the portable code, you may want to use `base64->bytestring` in `srfi.207` (see Section 11.47 [String-notated bytevectors], page 776).

**Base32**

RFC4648 defines two completely different encodings as base-32; The standard Base32 uses A–Z for 0 to 25, and 2–7 for 26 to 31. The “base32hex” variant uses 0–9 for 0 to 9, and A–V for 10 to 31. Since RFC states the latter should not be referred to as base32, we provide separate procedures for those encodings.

**base32-encode** :key omit-padding line-width  
base32hex-encode :key omit-padding line-width  
{rfc.base64} The procedures are port-based; they read input from the current standard port, and emits the base32 or base32hex encoded result to the current output port.

The keyword arguments `omit-padding` and `line-width` are the same as `base64-encode`.

(base32-encode-message "Hau'oli makahiki hou!")  
⇒ "JBQXKJ3PNRUSA3LBNFGW2LLNEQQGQ33VEE======"
(base32hex-encode-message "Hau'oli makahiki hou!")
⇒ "91GNA9RFHKI0RB1DDGMBB4G6GRRL44======"

base32-decode :key strict
base32hex-decode :key strict

{rfc.base64} Read Base32/Base32hex encoded string from the current input port, and emits
the decoded result to the current output port.

If a true value is given to the strict argument, an error is thrown if non-whitespace illegal
characters exist in the input, or the proper padding characters are missing. The default of
strict is #f and illegal characters are just ignored.

base32-decode-string-to target string :key strict
base32hex-decode-string-to target string :key strict

{rfc.base64} Decode a Base32/Base32hex encoded string
string, and returns the result as
an instance of class target, which must be either <u8vector> or <string>.

The meaning of strict argument is the same as in base32-decode.

(base32-decode-string-to <string> "JBQXKJ3PNRUSA3LBBNWQ2LLNEQGQ33VEE======")
⇒ "Hau'oli makahiki hou!"

Base16

Base16 is simply the hexadecimal encoding of input octets. RFC4648 defines to use uppercase
letters A-F for digits 10 to 15. In practice, lowercase letters are also used for hexadecimal
encoding, so we provide that option, too.

base16-encode :key lowercase
{rfc.base64} Read data from the current input port, and emits Base16 encoded result to
the current output port.

By default, A-F are used for digit 10 to 15, as specified in RFC4648. If lowercase argument
is given and true, a-f are used instead.

Note that no padding character is ever generated in Base16 encoding.

base16-encode-message message :key lowercase
{rfc.base64} Base16 encode message, which should be a string or a u8vector, and returns
the result in a string. The lowercase argument is the same as base16-encode.

(base16-encode-message "Aloha")
⇒ "416C6F6861"
(base16-encode-message "Aloha" :lowercase #t)
⇒ "416c6f6861"

base16-decode :key strict
{rfc.base64} Read Base16 encoded string from the current input port, and emits the de-
coded result to the current output port.

By default, characters other than Base16 digits are ignored, and both upperase and lowercase
alphabets are recognized. If strict argument is given and true, however, characters other than

base16-decode-string-to target string :key strict
{rfc.base64} Decode Base16-encoded string string, and returns the result as an instance of
class target, which must be either <u8vector> or <string>.

The strict argument is the same as base16-decode.

(base16-decode-string-to <u8vector> "416c6f6861")
⇒ #\u8(65 108 111 104 97)
(base16-decode-string-to <string> "416c6f6861")
⇒ "Aloha"

**Deprecated API**

The following API is deprecated. Avoid using them in new code.

- **base64-encode-string**
  
  string :key line-width digits url-safe

- **base64-encode-bytevector**
  
  u8vector :key line-width digits url-safe

  `{rfc.base64} Deprecated. Converts contents of string/u8vector to Base64 encoded format. For base64-encode-string, input string can be either complete or incomplete string; it is always interpreted as a byte sequence. The digits and url-safe arguments are the same as base64-encode.

- **base64-decode-string**
  
  string :key url-safe strict

- **base64-decode-bytevector**
  
  u8vector :key url-safe strict

  `{rfc.base64} Deprecated. Decodes a Base64 encoded string string and returns the result as a string or a u8vector, respectively. The conversion terminates at the end of string or the termination character (=).

  The characters which are not in legal Base64 encoded character set are silently ignored by default. If a true value is given to the strict keyword argument, however, non-whitespace out-of-set characters in the input raises an error.

  The digits and url-safe arguments are the same as base64-encode.

12.43 **rfc.cookie - HTTP cookie handling**

**rfc.cookie**

Defines a set of functions to parse and construct a “cookie” information defined in RFC 6265.

- **parse-cookie-string**
  
  string :optional version

  `{rfc.cookie} Parse a cookie string string, which is the value of “Cookie” request header. Usually, the same information is available to CGI program via the environment variable HTTP_COOKIE.

  If the cookie version is known, via “Cookie2” request header, the integer version must be passed to version. Otherwise, parse-cookie-string figures out the version from string.

  The result has the following format.

  ```
  ((<name> <value> [:path <path>] [:domain <domain>] [:port <port>])
   ...)
  ```

  where <name> is the attribute name, and <value> is the corresponding value. If the attribute doesn’t have value, <value> is #f. (Note that it differs from the attribute having null value, ".") If the attribute has path, domain or port options, it is given as a form of keyword-value pair.

  Note: To retrieve the value of a specific cookie conveniently, you can use rfc822-header-ref (see Section 12.41 [RFC822 message parsing], page 917).

- **construct-cookie-string**
  
  specs :optional version

  `{rfc.cookie} Given list of cookie specs, creates a cookie string suitable for Set-cookie2 or Set-cookie header.

  Optional version argument is accepted but ignored; it’s for the backward compatibility.

  Each cookie spec has the following format.

  ```
  (<name> <value> [:expires <data/time>] [:max-age <age>])
  ```
[:domain <domain>] [:path <path>]
[:secure <boolean>] [:http-only <boolean>]

Where,

<name> A string. Name of the cookie.

<value> Value of the cookie. May be a string, or \#f if no value is needed.

<domain> <path>
Strings.

<boolean>
Boolean value

<age>
Non-negative exact integers

<date/time>
Either an integer (seconds since Epoch), a formatted date string, <time> or <date> objects (see Section 6.24.9 [Time], page 316, see Section 11.7 [Time data types and procedures], page 701).

The order of attributes does not matter. The attribute values are quoted appropriately.

For the backward compatibility to the older specification (RFC2965, RFC2109, and old Netscape cookie specification), it recognizes a few more attributes (comment, comment-url, discard, port, and version). They are rendered to the output, but the new code should not use them.

Return value is a list of cookie strings, each of which stands for each cookie. For old-style protocol (using Set-cookie header) you must send each of them by individual header. For new-style protocol (using Set-cookie2 header), you can join them with comma and send it at once. See RFC6265 for further details.

Example:

(construct-cookie-string
  '(("name" "foo" :domain "foo.com" :path "/" :max-age 86400)))
⇒ ("name=foo;Domain=foo.com;Path=/;Max-age=86400")

12.44 rfc.ftp - FTP client

rfc.ftp

This module provides a set of convenient functions to access ftp servers.

<ftp-connection>
{rfc.ftp} An object to keep FTP connection to a server. It has the following public slots.

transfer-type [Instance Variable of <ftp-connection>]
FTP transfer type. Must be one of the following symbols: ascii, binary (default), and image.

passive [Instance Variable of <ftp-connection>]
True if the client uses passive connection.

log-drain [Instance Variable of <ftp-connection>]
This slot must hold a <log-drain> instance (see Section 9.17 [User-level logging], page 458) or \#f. If it has a <log-drain> instance, ftp communication logs are put to it.
<ftp-error>  
{rfc.ftp} This type of exception is thrown when the ftp server returns an error code. Inherits <error>. The message field contains the server reply, including the status code.

call-with-ftp-connection host proc :key passive port username
password account log-drain  
{rfc.ftp} A high-level convenience routine to open an ftp connection to an ftp server and calls the given procedure.

The server is specified by host. Optionally, you can add user name and/or port number by the form user@servername:port. If present, user and port portion in host supersedes the keyword arguments.

If ftp connection to host is established successfully, proc is called with one argument, which is an instance of <ftp-connection>. When proc returns, the connection is closed and the return value(s) of proc is/are returned from call-with-ftp-connection. When an exception is thrown, the ftp connection is closed before the exception escapes from call-with-ftp-connection.

When a true value is given to the keyword argument passive, created ftp connection will use passive mode to send/receive data. The default is the active mode.

The keyword argument port, username, and password specify the port number, username, and password, respectively. When omitted, the port number defaults to 21, username to "anonymous", and password to "anonymous@". Note that the port number and/or username are ignored when those information is given in the host argument.

If the keyword argument account is given, its value is passed to ftp ACCT command when requested by the server at login time. The default value is a null string "".

The keyword argument log-drain is set to the created ftp connection’s log-drain slot.

ftp-transfer-type conn  
{rfc.ftp} Returns the transfer type of the ftp connection conn. Can be used with setter, e.g. (set! (ftp-transfer-type conn) ’ascii).

ftp-passive? conn  
{rfc.ftp} Returns true iff ftp connection uses passive data retrieval.

ftp-login host :key passive port username password account log-drain  
{rfc.ftp} Connects to the ftp server specified by host, authenticate the user, and returns a newly created <ftp-connection> instance. This procedure is called implicitly when you use call-with-ftp-connection. The semantics of the host argument and the keyword arguments are the same as call-with-ftp-connection.

ftp-quit conn  
{rfc.ftp} Sends ftp QUIT command to the connection conn and shutdown the connection.

Once a connection is shut down, you cannot communicate through this connection.

ftp-chdir conn dirname  
{rfc.ftp} Changes the remote directory to dirname.

ftp-remove conn path  
{rfc.ftp} Removes the remote file named by path.

ftp-help conn :optional option . . .  
{rfc.ftp} Sends ftp HELP commands. Options must be strings, and will be passed to the HELP command arguments.
ftp-mkdir conn dirname
{rfc.ftp} Creates a directory dirname. Returns the created directory name.

ftp-current-directory conn
{rfc.ftp} Returns the current remote directory.

ftp-site conn arg
{rfc.ftp} Sends ftp SITE command with the argument arg. The SITE command’s semantics depends on the server. Returns the server reply.

ftp-rmdir conn dirname
{rfc.ftp} Removes remote directory specified by dirname. Returns the server reply.

ftp-stat conn :optional pathname
{rfc.ftp} Sends ftp STAT command to the server. RFC959 defines several different semantics of this command. See RFC959 for the details. Returns the server reply.

ftp-system conn
{rfc.ftp} Queries the server’s operating system by ftp SYST command. Returns the server reply without status code.

(call-with-ftp-connection "localhost" ftp-system)
⇒ "UNIX Type: L8"

ftp-size conn path
{rfc.ftp} Queries the size of the remote file specified by path. Returns the integer value. Note: The size may differ whether the connection is in ascii mode or binary mode; furthermore, some ftp server may returns the value only if the connection is in binary mode. Make sure you have desired transfer type in the connection.

ftp-mdtm conn path
{rfc.ftp} Queries the modification time of the remote file specified by path. This function returns the server’s reply as is, including the status code. Use ftp-mtime below to obtain a parsed result.

ftp-mtime conn path :optional local-time?
{rfc.ftp} Queries the modification time of the remote file specified by path, and returns the result in a <date> object (see Section 11.7 [Time data types and procedures], page 701). If a true value is given to local-time?, the returned date is in local time. Otherwise, the returned date is in UTC.

ftp-noop conn
{rfc.ftp} Sends ftp NOOP command and returns the server’s reply.

ftp-list conn :optional path
{rfc.ftp} Returns the information about the files within the remote file or directory specified by path, or the current remote directory, much like ls(1) format. Returns a list of strings, where each string is for each line of the server’s reply. The exact format depends on the server.

ftp-name-list conn :optional path
ftp-ls conn :optional path
{rfc.ftp} Return the list of names in the specified path, or the current remote directory, without any other information. ftp-ls is just an alias of ftp-name-list for the convenience. Note that the server may return an error if there’s no files in the remote directory.
ftp-get conn path :key sink flusher
{rfc.ftp} Retrieves a remote file path. The retrieved data is sent to an output port given to sink. Once all the data is retrieved, a procedure given to flusher is called with the port sink as an argument, and its return value(s) is/are returned from ftp-get.

The default values of sink and flusher are a newly created string port and get-output-string, respectively. That is, ftp-get returns the retrieved data as a string by default. You don’t want this behavior if the retrieved file is huge.

ftp-put conn from-file :optional to-file
{rfc.ftp} Sends the local file specified by from-file to the remote server as the name specified by to-file. If to-file is omitted, the basename of from-file is used. Returns the server response.

ftp-put-unique conn from-file
{rfc.ftp} Sends the local file specified by from-file to the remote server. The remote side filename is guaranteed to be unique. Returns two values—the final server response, and the remote file name. The second value can be #f if the remote host doesn’t support RFC1123 (which must be rare).

ftp-rename conn from-name to-name
{rfc.ftp} Renames the remote file specified by from-name to the name to-name. Returns the final response of the server.

12.45 rfc.hmac - HMAC keyed-hashing

rfc.hmac
This module implements HMAC algorithm, Keyed-hashing for message authentication, defined in RFC 2104.

For simple batched keyed hashing, you can use high-level API hmac-digest and hmac-digest-string. Or you can create <hmac> object and update its state as the data coming in.

Following is a typical way to digest a given string.

(use rfc.sha) ; import desired digest algorithm
(use rfc.hmac)

(hmac-message <sha256> "secret-key" "payload")
⇒ #u8(16 170 46 28 37 56 70 79 247 95 6 71 39 30 59 167 70 188 163
252 222 175 50 44 88 27 245 133 30 140 221 183)

(hmac-message-to 'hex <sha256> "secret-key" "payload")
⇒ "10aa2e1c2538464ff75f0647271e3ba746bca3fcdeaf322c581bf5851e8cddb7"

High-level API

hmac-to target algorithm key
{rfc.hmac} Read input data from the current input port, and compute HMAC digest with a hash algorithm hash-class and a key key.

The result is returned as an instance of class target, which can be <u8vector> or <string>. Some symbols is also allowed to specify encoding: base64, base64url, base64url-nopad, base32, base32hex, base16, and hex. See Section 12.82 [Message digester framework], page 1017, for the details of those encodings.

The hash-class argument must be a class implementing <message-digest-algorithm>, such as <sha256> of rfc.sha (see Section 12.53 [SHA message digest], page 946).

The key key may be a string or a u8vector.
**hmac-message-to target hash-class key message**  
{rfc.hmac} Returns an HMAC digest of message, computed with a hash algorithm hash-class, with a key key.

The hash-class argument must be a class inheriting <message-digest-algorithm>, e.g. <sha256> class from rfc.sha (see Section 12.53 [SHA message digest], page 946).

You can pass both a string and a u8vector to key/message.

The target can be a class in which the result is returned, either <u8vector> or <string>, or a symbol specifying the encoding: base64, base64url, base64url-nopad, base32, base32hex, base16, and hex. See Section 12.82 [Message digester framework], page 1017, for the details of those encodings.

**hmac-message hash-class key message**  
{rfc.hmac} A shortcut of (hmac-message-to <u8vector> hash-class key message).

```scheme
(hmac-message <sha256> "secret" "Aloha, honua")  
⇒ #u8(78 232 135 13 246 139 36 242 133 33 99 185 4 249 244 22 243 46 120 130 192 235 60 187 172 180 223 169 179 247 186 185)
```

**hmac-verify hash-class digest key message**  
{rfc.hmac} Verify message by comparing its digest with the given digest. Functionally it is (equal? digest (hmac-message hash-class key message)), but the digests are compared in a way that can avoid timing attack.

Hash-class is a class inherits <message-digest-algorithm> such as <sha256>. Digest must be a u8vector. Either a string or a u8vector is allowed for each of key and message.

Returns #t if digest is valid digest for the message, #f if not.

```scheme
(hmac-verify <sha256>  
  '#u8(78 232 135 13 246 139 36 242 133 33 99 185 4 249 244 22 243 46 120 130 192 235 60 187 172 180 223 169 179 247 186 185)  
  "secret"  
  "Aloha, honua")  
⇒ #t
```

**Low-level API**

**<hmac>**  
{rfc.hmac} Keeps state information of HMAC algorithm. This should not be directly instantiated; use make-hmac below.

**make-hmac algorithm key**  
{rfc.hmac} Creates and returns a new <hmac> object. The algorithm argument is a class inheriting <message-digest-algorithm> to specify digest algorithm, for example, <sha256> (see Section 12.53 [SHA message digest], page 946, for the details). The key argument is a key, either a string or a u8vector.

When you use high-level APIs such as hmac-message, you don’t need to make an <hmac> object since it is implicitly created. You only need to create it explicitly to use the following streaming update procedures.

**hmac-update! (hmac <hmac>) data**  
{rfc.hmac} Updates the internal state of hmac by data, which must be represented by a (possibly incomplete) string.

**hmac-final! (hmac <hmac>) optional target**  
{rfc.hmac} Finalizes the internal state of hmac and returns the digest in an instance of target, which must be a <string> or <u8vector> class metaobject, or a symbol to specify
encoding. See hmac-to above for the details of target. The default is <string>, for a historical reason.

Once finalized, you can’t call hmac-update! or hmac-final! on hmac.

**Deprecated API**

The following procedures are deprecated. You can use hmac-to and hmac-message instead.

hmac-digest :key key hasher [Function]
{rfc.hmac} Deprecated. Creates an <hmac> object and hash the data stream from the current input port, then returns the hashed result in an incomplete string.

hmac-digest-string string :key key hasher [Function]
{rfc.hmac} Deprecated. Creates an <hmac> object and hash the data in string, then returns the hashed result in an incomplete string.

**12.46 rfc.http - HTTP client**

rfc.http [Module]
This module provides a simple client API for HTTP/1.1, defined in RFC2616, "Hypertext Transfer Protocol – HTTP/1.1" (https://www.ietf.org/rfc/rfc2616.txt).

Current API implements only a part of HTTP/1.1 protocol. Support for some advanced features such as persistent connection may be added in the future versions.

If you’re looking a library to write HTTP server, see Gauche-makiki (https://github.com/shirok/Gauche-makiki).

<http-error> [Condition Type]
{rfc.http} This type of condition is raised when the server terminates connection prematurely or server’s response has invalid header fields. Inherits <error>.

**12.46.1 Http client mid-level API**

http-get server request-uri :key sink flusher redirect-handler secure . . . [Function]
http-head server request-uri :key redirect-handler secure . . . [Function]
http-post server request-uri body :key sink flusher redirect-handler secure . . . [Function]
http-put server request-uri body :key sink flusher redirect-handler secure . . . [Function]
http-delete server request-uri :key sink flusher redirect-handler secure . . . [Function]
{rfc.http} Send http GET, HEAD, POST, PUT and DELETE requests to the http server, respectively, and returns the server’s reply.

By default, if the server returns 300, 301, 302, 303, 305 and 307 status, these procedures attempts to fetch the redirected URL by the "location" reply message header if it is allowed by RFC2616. This behavior can be turned off or customized by the redirect-handler keyword argument; see the "keyword arguments" heading below for the details.

**Required arguments:** The server argument specifies http server name in a string. A server name can be optionally followed by colon and a port number. You can use IP address, too; for IPv6, you have to surround the address in brackets.

Additionally, you can specify "unix:/path" where /path is the absolute path to the unix domain socket; this allows to connect to httpd listening on unix domain sockets. Examples: "w3c.org", "mycompany.com:8080", "192.168.0.1:8000", "[::1]:8000"
The request-uri argument can be a string or a list. If it is a string, it’s request-uri specified in RFC2616; usually, this is the path part of http url. The string is passed to the server as is, so the caller must properly convert character encodings and perform necessary url encodings.

If request-uri is a list, it must be in the following form:

\[
(path \ (name \ value) \ ...)\]

Here, path is a string specifying up to the path component of the request-uri. From provided alist of names and values, http procedures compose a query string in application/x-www-form-urlencoded format as defined in HTML4, and append it to path. For example, the following two requests have the same effect. Note that url escaping is automatically handled in the second call.

\[
(http-get "example.com" "/search?q=foo%20bar&n=20")\]

\[
(http-get "example.com" '("/search" (q "foo bar") (n 20)))\]

If request-encoding keyword argument is also given, names and values are converted into the specified character encoding before url escaping. If it is omitted, gauche’s internal character encoding is used.

Some procedures take the third argument, body, to specify the body of the request message. It can be a string, which will be copied verbatim to the request body, or a list, which will be encoded in multipart/form-data message.

If body is a list, it is a list of parameter specs. Each parameter spec is either a list of name and value, e.g. ("submit" "OK") or a name followed by keyword-value list, e.g. ("upload" :file "logo.png" :content-type "image/png"). The first form is for the convenience. It is also compatible to the query parameter list in request-uri, so that you can use the same format for GET and POST request. Each value is put in a MIME part with text/plain media type, with the character encoding specified by request-encoding keyword argument described below.

The second form allows further control over each MIME part’s attributes. The following keywords are treated specially.

: value Specifies the value of the parameter. The convenience form, (name val), is just an abbreviation of (name :value val).

: file Specifies the pathname of the file, whose content is inserted as the value of the parameter. Useful to upload a file. This option has precedence over :value. MIME type of the part is set to application/octet-stream unless specified otherwise.

: content-type Overrides the MIME type of the part. A charset parameter is added to the content-type if not given in this argument.

: content-transfer-encoding Specifies the value of content-transfer-encoding; currently the following values are supported: 7bit, binary, quoted-printable and base64. If omitted, binary is used.

Other keywords are used as the header of the MIME part.

Return values: All procedures return three values.

The first value is the status code defined in RFC2616 in a string (such as "200" for success, "404" for "not found").

The second value is a list of parsed headers—each element of list is a list of (header-name value ...), where header-name is a string name of the header (such as "content-type" or
"location"), and value is the corresponding value in a string. The header name is converted to lowercase letters. The value is untouched except that "soft line breaks" are removed, as defined in RFC2822. If the server returns more than one headers with the same name, their values are consolidated to one list. Except that, the order of the header list in the second return value is the same as the order in the server’s reply.

The third value is for the message body of the server’s reply. By default, it is a message body itself in a string. If the server’s reply doesn’t have a body, the third value is #f. You can change how the message body is handled by keyword arguments; for example, you can directly store the returned message body to a file without creating intermediate string. The details are explained below.

**Keyword arguments:** By default, these procedures only attaches "Host" header field to the request message. You can give keyword arguments to add more header fields.

```scheme
(http-get "foo.bar.com" "/index.html"
 :accept-language "ja"
 :user-agent "My Scheme Program/1.0")
```

The following keyword arguments are recognized by the procedure and do not appear in the request headers.

- **request-encoding**
  When a list is given to the request-uri or body arguments, the characters in names and values of the parameters are first converted to the character encoding specified by this keyword argument, then encoded into application/x-www-form-urlencoded or multipart/form-data MIME formats. If this argument is omitted, Gauche’s internal character encoding is used.

  For multipart/form-data, you can override character encodings for individual parameters by giving content-type header. See the description of body arguments above.

  If you give a string to request-uri or body, it is used without encoding conversion.

  It is caller’s responsibility to ensure desired character encodings are used.

- **proxy**
  Specify http proxy server in a string of a form hostname or hostname:port. If omitted, the value of the parameter http-proxy is used.

- **redirect-handler**
  Specifies how the redirection is handled when the server responds with 3xx status code. You can pass #f, #t or a procedure. The default is #t.

  If #f is given, no redirect attempt will be made; the 3xx status code and response is just returned from http-* procedures as they are.

  If a procedure is given, it is called when the response status code is 3xx. The procedure takes four arguments, the request method (in symbol, e.g. GET), the response status code (in string, e.g. "302"), the parsed response headers and the response body (a string if there’s a body, or #f if the response doesn’t have a body).

  The procedure can return a pair or #f. If it is a pair, it should be (method . url), where method is a symbol (e.g. GET) and url is a string representing url.

  If a pair is returned, the http-* procedures tries to send the request with the given method (it allows a redirection of POST request to be GET, for example).

  If it is #f, no further attempt of redirection is made.

  If redirect-handler is #t, which is the default, then it works as if the value of the parameter http-default-redirect-handler is passed to redirect-handler. The parameter contains a procedure with reasonable default behavior. See the http-default-redirect-handler entry below for the details.
A loop in redirection is detected automatically and `<http-error>` is thrown.

**no-redirect**
This is an obsoleted keyword argument kept only for the backward compatibility. If a true value is given, it has the same effect as specifying `#f` to `redirect-handler`.

**secure**
If a true value is given, the secure connection is used. The value specifies the secure transport agent to establish https connection. It can be `#t` or a symbol `tls` or `stunnel`. If `#f` is given (default), non-secure plain http is used. See the “Secure connection” section below.

**auth-user, auth-password**
If given, the authorization header using Basic Authentication (RFC2617) is added to the request. In future, we might add support for other authentication scheme.

**sink, flusher**
You can customize how the reply message body is handled by these keyword arguments. You have to pass an output port to `sink`, and a procedure that takes two arguments to `flusher`.

When the procedure starts receiving the message body, it feeds the received chunk to `sink`. When the procedure receives entire message body, `flusher` method is called with `sink` and a list of message header fields (in the same format to be returned in the second value from the procedure). The return value of `flusher` becomes the third return value from the procedure.

So, the default value of `sink` is a newly opened string port and the default value of `flusher` is `(lambda (sink headers) (get-output-string sink))`.

The following example saves the message body directly to a file, without allocating (potentially very big) string buffer.

```lisp
(call-with-output-file "page.html"
 (lambda (out)
  (http-get "www.schemers.org" "/"
    :sink out :flusher (lambda _ #t))))
```

### 12.46.2 Http client utilities

The module also provides some utility procedures.

**http-user-agent :optional value**

{rfc.http} The value of this parameter is used as a default value to pass to the user-agent header. The default value is something like `gauche.http/*`, where `*` is Gauche’s version. An application is encouraged to set this parameter appropriately.

**http-proxy :optional value**

{rfc.http} This value is used as the default http proxy name by `http-get` etc. The default value is `#f` (no proxy).

**http-default-redirect-handler :optional value**

{rfc.http} Specifies the behavior of redirection if no `redirect-handler` keyword argument is given to the `http-*` procedures. If you change this value, it must be a procedure that follows the protocol of `redirect-handler`; see the description of `http-*` procedures above.

The default behavior is as follows:

300, 301, 305, 307
Redirect to the url given to the `location` header only if the original request method is `GET` or `HEAD`. 
302 Redirect to the url given to the `location` header. If the original request method is `HEAD`, it is used again. Otherwise, `GET` method is used. Strictly speaking, this is a violation of RFC2616. However, as the note in RFC2616 says, many user agent do this, so we follow the flock. (We may change this in future.)

303 Redirect to the url given to the `location` header. If the original request method is `HEAD`, it is used again. Otherwise, `GET` method is used. other than above

No redirection is made.

The following code is an example of intercepting the default behavior in a specific request:

```
(http-get server uri :redirect-handler
 (^
 [method status headers body]
 (if (and (equal? status "302")
 (not (member method '(GET HEAD))))
 #f
 ((http-default-request-handler) method status headers body))))
```

**http-compose-query**

`path` `params` :optional encoding

{rfc.http} A helper procedure to create a request-uri from a list of query parameters. 

The format of `params` argument is the same as the list format of `body` argument of http request procedures. The result is written to an output port `port`, and the boundary string used to compose MIME message is returned. Alternatively you can pass `#f` to the `port` to get the result in a string. In that case, two values are returned, the MIME message string and the boundary string.

**Encoding** specifies the character encodings to be used. When omitted, Gauche’s native encoding is used.

```
(define p (open-output-string))

(http-compose-form-data #f
 (http-compose-query "path" params :optional encoding)
 (http-compose-query "" '')
 ⇒ "x=a%20b&x=2"

(http-compose-query "" '')
 ⇒ "x=a%20b&x=2"
```

**http-compose-form-data**

`params` `port` :optional encoding

{rfc.http} A helper procedure to create multipart/form-data from a list of parameters. 

The format of `params` argument is the same as the list format of `body` argument of http request procedures. The result is written to an output port `port`, and the boundary string used to compose MIME message is returned. Alternatively you can pass `#f` to the `port` to get the result in a string. In that case, two values are returned, the MIME message string and the boundary string.

**Encoding** specifies the character encodings to be used. When omitted, Gauche’s native encoding is used.

```
(define p (open-output-string))

(http-compose-form-data #f
 (http-compose-query "path" params :optional encoding)
 (http-compose-query "" '')
 ⇒ "boundary-fh87o52rp6zkubp2uhdmo"

(get-output-string p)
 ⇒
```
Chapter 12: Library modules - Utilities

12.46.3 Secure http connection

When you pass a true value to secure keyword argument, the request-making APIs such as http-get use a secure connection. That is, it connects with https instead of http. The actual value for the keyword argument can be one of the followings:

- #t: The rfc.tls module is used for the secure connection. See Section 12.54 [Transport layer security], page 946, for the details—you might need to set CA certificate bundle path.
- tls: The rfc.tls module is used for the secure connection. See Section 12.54 [Transport layer security], page 946, for the details—you might need to set CA certificate bundle path.
- stunnel: The external process stunnel is spawned and used for the secure connection.
- #f: Secure connection is not used.

If specified secure connection subsystem isn’t available in the running Gauche, an error is signaled. Use the following procedure to check if you can use secure connections:

**Function**

```
http-secure-connection-available? :optional type
```

Returns #t if running Gauche can use secure connection of the given type, #f otherwise.

12.47 rfc.icmp - ICMP packets

**Module**

```
rfc.icmp
```

This module provides some basic utilities to construct and parse ICMP packets.

For the functions below, buffer should be a writable u8vector of the enough size.

Parsing functions takes offset as well as buffer, which specifies the beginning of the ICMP packet. Using the offset you can carry the whole IP packet in buffer, without creating a new buffer to extract ICMP portion.

**Function**

```
icmp4-fill-echo! buffer ident sequence data
```

Fills buffer with the ICMPv4 Echo Request packet. Data must be a u8vector. The checksum field is left to be zero, which can be filled by icmp4-fill-checksum!.

**Function**

```
icmp4-fill-checksum! buffer size
```

Calculates the ICMPv4 checksum of the packet in the buffer, of size length (the size of the packet, not the buffer), and fills the checksum field of the packet.

**Function**

```
icmp6-fill-echo! buffer ident sequence data
```

Fills buffer with the ICMPv6 Echo Request packet. Data must be a u8vector. The checksum field is left to be zero, which is to be filled by the kernel (so you don’t need to fill by yourself).
12.48 rfc.ip - IP packets

rfc.ip

This module provides some basic utilities to parse raw IP packets.

The packet argument in the following functions must be any type of uniform vector (see Section 6.13.2 [Uniform vectors], page 198), containing a raw IP packet including its IP header. Those functions work for both IPv4 and IPv6 packets; however, reading from a raw IPv6 socket returns a packet without IPv6 header, so you usually don’t need to use these functions.

The offset argument specifies the beginning of the IP packet in packet. If packet contains only one IP packet you can pass 0. It is not an optional argument, since these routines may be used in speed-sensitive inner loop.

ip-version packet offset
    {rfc.ip} Returns the IP version number (either 4 or 6) of the given IP packet.

ip-header-length packet offset
    {rfc.ip} Returns the size of IP header of the given packet in octets, including any IP header options.

ip-protocol packet offset
    {rfc.ip} Returns the IP protocol number of the given packet.

ip-source-address packet offset
    {rfc.ip} Returns the source address in the given packet in an integer, respectively.

ip-destination-address packet offset
    {rfc.ip} Returns the destination address in the given packet in an integer, respectively.
12.49 rfc.json - JSON parsing and construction

rfc.json

Procedures to parse JSON (RFC7159) data to S-expressions, and convert S-expressions to JSON representation, are provided.

Parsing JSON

<json-parse-error>

The parser parse-json and parse-json-string raise this condition when they encounter invalid JSON syntax. It inherits <error>, and adds the following slot.

position

The input position, counted in characters, where the error occurred.

json-nesting-depth-limit

Its value must be a real number, specifying the maximum nesting depth of JSON text that can be parsed by parse-json. If the input exceeds the value, an <json-parse-error> is thrown. The default value is +inf.0.

parse-json :optional input-port

Reads and parses the JSON representation from input-port (default is the current input port), and returns the result in an S-expression. May raise a <json-parse-error> condition when parse error occurs, or the nesting level exceeds the value of json-nesting-depth-limit.

The following table shows how JSON datatypes are mapped to Scheme objects.

true, false, null
Symbols true, false and null. (Customizable by json-special-handler)

Arrays Scheme vectors. (Customizable by json-array-handler)

Objects Scheme assoc-lists, in which keys are strings, and values are Scheme objects. (Customizable by json-object-handler)

Numbers Scheme inexact real numbers.

Strings Scheme strings.

Since the parser used internally in parse-json prefetches characters, some characters after the parsed JSON expression may already been read from port when parse-json returns. That is, you cannot call parse-json repeatedly on port to read subsequent JSON expressions. Use parse-json* if you need to read multiple JSON expressions.

parse-json* :optional input-port

Read JSON repeatedly from input-port until it reaches EOF, and returns parsed results as a list.

parse-json-string str

Parses the JSON string and returns the result in an S-expression. May raise a <json-parse-error> condition when parse error occurs.

See parse-json above for the mappings from JSON datatypes to Scheme types.

json-array-handler

The value of these parameters must be a procedure that takes one argument: for json-array-handler, it is a list of elements of a JSON array, for json-object-handler, it
is a list of conses of key and value of a JSON object, and for \texttt{json-special-handler}, it is one of the symbols \texttt{false}, \texttt{true} or \texttt{null}.

Whenever \texttt{parse-json} reads a JSON array, a JSON object, or one of those special values, it calls corresponding parameter to get a Scheme object.

The default value of these parameters are \texttt{list->vector}, \texttt{identity}, and \texttt{identity}, respectively.

The following example maps JSON objects to hash tables.

\begin{verbatim}
(parameterize ([json-object-handler (cut alist->hash-table <> 'string=?)])
  (parse-json-string "{\"a\":1, \"b\":2\"}")
⇒ #<hash-table ...>
\end{verbatim}

\section*{Constructing JSON}

\texttt{<json-construct-error>}  
\hspace{2em} [Condition type]  
\hspace{2em} \{rfc.json\} The converters \texttt{construct-json} and \texttt{construct-json-string} raise this condition when they cannot convert given Scheme object to JSON. It inherits \texttt{<error>}, and adds the following slot.

\begin{itemize}
  \item \texttt{object} [Instance Variable of \texttt{<json-construct-error>}]  
    The Scheme object that cannot convert to JSON representation.
\end{itemize}

\texttt{<json-mixin>}  
\hspace{2em} [Class]  
\hspace{2em} \{rfc.json\} A Gauche object of a class inheriting this mixin class is automatically serialized to JSON object with the following rule:

\begin{itemize}
  \item Each slot of the object is examined. Only the slots with its slot definition has a true value in \texttt{:json-name} option are the subject to be serialized.
  \item If \texttt{:json-name} option is \#t, the slot name is used as the key; otherwise, string representation (\texttt{x->string}) of the option’s value is used as the key.
  \item Then the slot value is serialized recursively.
\end{itemize}

\begin{verbatim}
(define-class <foo> (<json-mixin>)
  ((slot-1 :init-keyword :slot-1 :json-name "Slot1")
   (slot-2 :init-keyword :slot-2 :json-name #t)
   (slot-3 :init-keyword :slot-3)))

(construct-json
  (make <foo> :slot-1 "xyz" :slot-2 123 :slot-3 'abc))
⇒ prints
  {"Slot1":"xyz","slot-2":123}
\end{verbatim}

\texttt{construct-json obj :optional output-port} \hspace{2em} [Function]  
\texttt{construct-json-string obj} \hspace{2em} [Function]  
\hspace{2em} \{rfc.json\} Creates JSON representation of Scheme object \texttt{obj}. \texttt{construct-json} writes out the result to \texttt{output-port}, whose default is the current output port. \texttt{construct-json-string} returns the result in a string.

If \texttt{obj} contains a Scheme object that cannot be mapped to JSON representation, a \texttt{<json-construct-error>} condition is raised.

Scheme objects are mapped to JSON as follows:

\begin{verbatim}
symbol false, #f
  false

symbol true, #t
  true
\end{verbatim}
symbol null
null

list, instance of "<dictionary>"
JSON object (list must be an assoc list of key and value).

string string
real number number

instance of "<sequence>" (except strings and lists)
JSON array

12.50 rfc.md5 - MD5 message digest

rfc.md5 [Module]
This module implements MD5 message digest algorithm, defined in RFC 1321 (https://www.ietf.org/rfc/rfc1321.txt). The module extends util.digest (see Section 12.82 [Message digester framework], page 1017).

Typically, all you need to do is to pass the class object <md5> to the message digester; see Section 12.82 [Message digester framework], page 1017, and also see Section 12.45 [HMAC keyed-hashing], page 928.

<md5> [Class]
{rfc.md5} The instance of this class keeps internal state of MD5 digest algorithm. This class implements util.digest framework interface, so you can pass this class to message digest procedures such as digest-message-to (see Section 12.82 [Message digester framework], page 1017).

The following procedures are deprecated. Use generic message digester (see Section 12.82 [Message digester framework], page 1017) or HMAC keyed digester (see Section 12.45 [HMAC keyed-hashing], page 928).

md5-digest [Function]
{rfc.md5} Deprecated. Reads data from the current input port until EOF, and returns its digest in an incomplete string.

md5-digest-string string [Function]
{rfc.md5} Deprecated. Digest the data in string, and returns the result in an incomplete string.

12.51 rfc.mime - MIME message handling

rfc.mime [Module]
This module provides utility procedures to handle Multipurpose Internet Mail Extensions (MIME) messages, defined in RFC2045 thorough RFC2049. Provided APIs include procedures to parse or compose MIME-specific header fields, and parse or compose MIME-encoded message bodies.

This module mainly focuses on providing low-level building-block procedures, on top of which application-specific modules are to be built. For example, rfc.http uses this module to compose multipart/form-data message for the body of POST requests (see Section 12.46 [HTTP client], page 930).

This module is supposed to be used with rfc.822 module (see Section 12.41 [RFC822 message parsing], page 917).
Utilities for header fields

A few utility procedures to parse and generate MIME-specific header fields.

**mime-parse-version field**

[Function]
{rfc.mime} If `field` is a valid header field for MIME-Version, returns its major and minor versions in a list. Otherwise, returns `#f`. It is allowed to pass `#f` to `field`, so that you can directly pass the result of `rfc822-header-ref` to it. Given parsed header list by `rfc822-read-headers`, you can get mime version (currently, it should be `(1 0)`) by the following code.

```
(mime-parse-version (rfc822-header-ref headers "mime-version"))
```

Note: simple regexp such as `/\d+\.\d+/` doesn't do this job, for `field` may contain comments between tokens.

**mime-parse-content-type field**

[Function]
{rfc.mime} Parses the "content-type" header field, and returns a list such as:

```
(type subtype (attribute . value) ...)
```

where `type` and `subtype` are MIME media type and subtype in a string, respectively. If `field` is not a valid content-type field, `#f` is returned.

```
(mime-parse-content-type "text/html; charset=iso-2022-jp")
⇒ ("text" "html" ("charset" . "iso-2022-jp"))
```

**mime-parse-content-disposition field**

[Function]
{rfc.mime} Parses Content-disposition header field as specified in RFC2183.

```
```

**mime-parse-parameters :optional iport**

[Function]
{rfc.mime} These are low-level utility procedures to parse and compose parameter part of header fields (as appeared in RFC2045 Section 5.1 etc).

Mime-parse-parameters reads the parameter part of the header body from an input port `iport`, and returns an assoc list of the parameter names and values. Conversely, mime-compose-parameters takes an assoc list of names and values, compose parameter part and emit it to `oport`. When omitted, the current input port and the current output port are used for `iport` and `oport`, respectively. You can pass `#f` to `oport` and mime-compose-parameters returns the result in a string instead of emitting it to a port.

```
(call-with-input-string
   "; name=foo; filename="foo/bar/baz"
   mime-parse-parameters)
⇒ ("name" . "foo") ("filename" . "foo/bar/baz")
```

```
(mime-compose-parameters
   '((("name" . "foo") ("filename" . "foo/bar/baz"))
     #f)
⇒ "; name=foo; filename="foo/bar/baz"
```

Mime-compose-parameters tries to insert folding line breaks between parameters to avoid the header line becomes too long. You can pass the beginning column position of the parameter part via `start-column` argument.

We plan to make these procedures handle RFC2231’s parameter value extension transparently in future.
mime-decode-word word
{rfc.mime} Decodes RFC2047-encoded word. If word isn’t an encoded word, it is returned as is.

Note that this procedure decodes only if the entire word is an “encoded word” defined in RFC2047. If you are dealing with a field that may contain multiple encoded word and/or unencoded parts, use mime-decode-text below.

(mime-decode-word "=?iso-8859-1?q?this=20is=20some=20text?=")
⇒ "this is some text"

mime-decode-text text
{rfc.mime} Returns a string in which all encoded words contained within text are decoded. This procedure can deal with a header field body that may contain mixture of non-encoded and encoded parts, and/or multiple encoded parts. One of such header field is the Subject field of email.

(mime-decode-text "This is =?US-ASCII?q?some=20text?=")
⇒ "This is some text"

Care should be taken if you apply this procedure to a “structured” header field body (see RFC2822 section 2.2.2). The proper way of parsing a structured header field body is to tokenize it first, then to decode each word using mime-decode-word. since the decoded text may contain characters that affects the tokenization. (However, if you can just show the header field in human readable way for informational purposes, you may just use mime-decode-text on entire header field for the convenience).

mime-encode-word word :key charset transfer-encoding
{rfc.mime} Encodes word in the RFC2047 format. The keyword argument charset specifies the character encoding scheme in string or symbol. whose default is utf-8. If charset differs from Gauche’s internal encoding and word is a complete string, the procedure converts the character encoding to charset, then performs transfer encoding.

(mime-encode-word "this is some text")
⇒ "=?utf-8?B?dGhpcyBpcyBzb21lIHRleHQ="

The keyword argument transfer-encoding specifies how the octets are encoded to transfer-safe characters. You can give a symbol b, B or base64 for Base64, and Q, q, quoted-printable for Quoted-printable transfer encodings. An error is raised if you pass values other than those. The default is Base64 encoding.

This procedure does not consider the length of the resulting encoded word, which RFC2047 recommends to be less than 75 octets. Use mime-encode-text below to conform the line length limit.

(Note: In most Gauche procedures, a keyword argument encoding is used to specify character encodings. In this context we have two encodings, however, and to avoid the confusion we chose to use the terms “charset” and “transfer-encoding” that appear in RFC documents.)

mime-encode-text text :key charset transfer-encoding line-width start-column force
{rfc.mime} Encode text in RFC2047 format if necessary, and considering line folding if the result gets too long.

The keyword arguments charset and transfer-encoding are the same as mime-encode-word. If the text only consists of printable ASCII characters, no encoding is done, and only line folding is considered. However, if a true value is given to the force argument, even ASCII-only text is encoded.

The line-width specifies the maximum line width of the result. Its default is 76. If the encoded word gets too long, it is splitted to multiple encoded words and CR LF SPC sequence (“folding
white space” defined in RFC2822) are inserted inbetween. You can suppress this behavior by passing #f or 0 to line-width. Since encoded word needs some overhead characters, it doesn’t make much sense to specify small value to line-width. Current implementation rejects line-width smaller than 30.

The start-column keyword argument can be used to shorten the first of folded lines to make room for header field name. For example, if you want to encode the body of a Subject header field, you can pass the value of (string-length "Subject: ") so that the encoded result can directly concatenated after the header field name. The default value is 0.

This procedure is not designed to encode parts of structured header fields, which have further restrictions such as which parts can be encoded and where the folding white spaces can be inserted. The robust way is to encode some parts first, then construct a structured header fields, considering line folding.

Streaming parser

The streaming parser is designed so that you can decide how to do with the message body before the entire message is read.

mime-parse-message port headers handler

{rfc.mime} The fundamental streaming parser. Port is an input port from where the message is read. Headers is a list of headers parsed by rfc822-read-headers; that is, this procedure is supposed to be called after the header part of the message is parsed from port:

(let* ((headers (rfc822-read-headers port)))
  (if (mime-parse-version (rfc822-header-ref headers "mime-version"))
      ;; parse MIME message
      (mime-parse-message port headers handler)
      ;; retrieve a non-MIME body
      ...
    ))

Mime-parse-message analyzes headers, and calls handler on each message body with two arguments:

(handler part-info xport)

Part-Info is a <mime-part> structure described below that encapsulates the information of this part of the message. Xport is an input port, initially points to the beginning of the body of message. The handler can read from the port as if it is reading from the original port. However, xport recognizes MIME boundary internally, and returns EOF when it reaches the end of the part. (Do not read from the original port directly, or it will mess up the internal state of vport).

Handler can read the part into the memory, or save it to the disk, or even discard the part. Whatever it does, it has to read from vport until it returns EOF.

The return value of handler will be set in the content slot of part-info. If the message has nested multipart messages, handler is called for each "leaf" part, in depth-first order. Handler can know its nesting level by examining part-info structure. The message doesn’t need to be a multipart type; if it is a MIME message type, handler is called on the body of enclosed message. If it is other media types such as text or application, handler is called on the (only) message body.

<mime-part>

{rfc.mime} A structure that encloses metainformation about a MIME part. It is constructed when the header of the part is read, and passed to the handler that reads the body of the part.

It has the following slots:
type [Instance Variable of <mime-part>]
MIME media type string. If content-type header is omitted to the part, an appropriate
default value is set.

subtype [Instance Variable of <mime-part>]
MIME media subtype string. If content-type header is omitted to the part, an appro-
priate default value is set.

parameters [Instance Variable of <mime-part>]
Associative list of parameters given to content-type header field.

transfer-encoding [Instance Variable of <mime-part>]
The value of content-transfer-encoding header field. If the header field is omitted, an
appropriate default value is set.

headers [Instance Variable of <mime-part>]
The list of header fields, as parsed by rfc822-read-headers.

parent [Instance Variable of <mime-part>]
If this is a part of multipart message or encapsulated message, points to the enclos-
ing part’s <mime-part> structure. Otherwise #f.

index [Instance Variable of <mime-part>]
Sequence number of this part within the same parent.

content [Instance Variable of <mime-part>]
If this part is multipart/* or message/* media type, this slot contains a list of parts within
it. Otherwise, the return value of handler is stored.

source [Instance Variable of <mime-part>]
This slot is only used when composing a MIME message. The caller can set this slot a
name of the file to be inserted into this part, instead of setting the entire content of the
file to the content slot. See mime-compose-message below for the more details.

mime-retrieve-body part-info xport outp
{rfc.mime} A procedure to retrieve message body. It is intended to to be a building block
of handler to be passed to mime-parse-message.

Part-info is a <mime-part> object. Xport is an input port passed to the handler, from which
the MIME part can be read. This procedure read from xport until it returns EOF. It also
looks at the transfer-encoding of part-info, and decodes the body accordingly; that is,
base64 encoding and quoted-printable encoding is handled. The result is written out to an
output port outp.

This procedure does not handle charset conversion. The caller must use CES conversion port
as outp (see Section 9.5 [Character code conversion], page 399) if desired.

A couple of convenience procedures are defined for typical cases on top of mime-retrieve-
body.

mime-body->string part-info xport
mime-body->file part-info xport filename
{rfc.mime} Reads in the body of mime message, decoding transfer encoding, and returns it
as a string or writes it to a file, respectively.
The simplest form of MIME message parser would be like this:

```lisp
(let ((headers (rfc822-read-headers port)))
  (mime-parse-message port headers
    (cut mime-body->string <> <>)))
```

This reads all the message on memory (i.e. the "leaf" <mime-part> objects' content field would hold the part’s body as a string), and returns the top <mime-part> object. Content transfer encoding is recognized and handled, but character set conversion isn’t done.

You may want to feed the message body to a file directly, or even want to skip some body according to mime media types and/or other header information. Then you can put the logic in the handler closure. That’s the reason that this module provides building blocks, instead of all-in-one procedure.

**Message composer**

```lisp
(rfc.mime) Composes a MIME multipart message. Mime-compose-message emits the result to an output port port, whose default is the current output port. Mime-compose-message-string makes the result into a string. You can give a boundary string via boundary argument; when omitted, a fresh boundary string is automatically generated by mime-make-boundary below.

Mime-compose-message returns the boundary string. Mime-compose-message-string returns two values, the result string and the boundary string.

The content of the message is provided by the parts argument, which can be a list of instances of <mime-part> (see above) or lists that describe parts. The list form is supported for the caller’s convenience, and internally it is converted to a list of <mime-part>s.

The syntax of each part element in parts are defined as follow.

```
<part> : <mime-part> | <mime-part-desc>
<mime-part> : an instance of the class <mime-part>
<mime-part-desc> : (<content-type> (<header> ...) <body>)
<content-type> : (<type> <subtype> <header-param> ...)
<header-param> : (<key> . <value>) ...
<header> : (<header-name> <encoded-header-value>)
     | (<header-name> (<header-value> <header-param> ...))
<body> : a string
     | (file <filename>)
     | (subparts <part> ...)
```

Note: In the first form of <header>, <encoded-header-value> must already be encoded using RFC2047 or RFC2231 if the original value contains non-ascii characters. In the second form, we plan to do RFC2231 encoding on behalf of the caller; but the current version does not implement it. The caller should not pass encoded words in this form, since it may result double-encoding when we implement the auto encoding feature; for the time being, the second form restricts ASCII-only values.

If <body> is a string, it is used as the part’s content. If <body> is (file filename), the content is read from the named file. If <body> is (subparts part ...), the part becomes nested MIME part.

It is the caller’s responsibility to give the proper content. For example, if <body> is in the third form, the part must have multipart content type.
The caller needs to provide proper `content-transfer-encoding` header, depending on the application. If none is given, the content is inserted into the message as is, which may be appropriate for some applications, but if you want to use the result in email message you certainly want to encode binary part with base64, for example.

```plaintext
mime-make-boundary

Returns a unique string that can be used as a boundary of a MIME multipart message.
```

### 12.52 rfc.quoted-printable - Quoted-printable encoding/decoding

```plaintext
rfc.quoted-printable

This module defines a few functions to encode/decode Quoted-printable format, defined in RFC 2045 (https://www.ietf.org/rfc/rfc2045.txt), section 6.7.
```

```plaintext
quoted-printable-encode :key line-width binary

Reads byte stream from the current input port, encodes it in Quoted-printable format and writes the result character stream to the current output port. The conversion ends when it reads EOF from the current input port. The keyword argument `line-width` specifies the maximum line width of the generated output in characters. If the encoded output creates a long line, the procedure inserts a "soft line break" so that each line is equal to or shorter than this number. Soft line breaks are removed when quoted-printable text is decoded. The default line width is 76. (The minimum meaningful number of line-width is 4). You can suppress soft line breaks by giving `#f` or `0` to `line-width`. By default, `quoted-printable-encode` generates CR-LF sequence for each line break in the input ("hard line break"). When a true value is given to the keyword argument `binary`, however, octets `#x0a` and `#x0d` in the input are encoded as `=0A` and `=0D`, respectively. See RFC2045 section 6.7 for the details.
```

```plaintext
quoted-printable-encode-message message :key line-width binary

Converts content of `message`, which can be either a string or a `u8vector`, to a string of Quoted-printable encoded format. The keyword arguments are the same as `quoted-printable-encode`.
```

```plaintext
quoted-printable-decode

Reads character stream from the current input port, decodes it from Quoted-printable format and writes the result byte stream to the current output port. The conversion ends when it reads EOF. If it encounters illegal character sequence (such as `=' followed by non-hexadecimal characters), it copies them literally to the output.
```

```plaintext
quoted-printable-decode-string-to target string

Decodes a Quoted-printable encoded string `string`, and returns the result as an instance of `target`, which must be either `<string>` or `<u8vector>`.
```

```plaintext
quoted-printable-encode-string string :key line-width binary

Deprecated. This is for the backward compatibility. Use `quoted-printable-encode-message` instead.
```

```plaintext
quoted-printable-decode-string string

Deprecated. This is for the backward compatibility. It is same as `(quoted-printable-decode-string-to <string> string)`.
```
12.53 rfc.sha - SHA message digest

rfc.sha [Module]
This module implements US Secure Hash Algorithm defined in RFC 4634. It provides SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 (the latter four are sometimes referred as SHA-2 collectively).

The module extends util.digest (see Section 12.82 [Message digester framework], page 1017). Typically, all you need to do is to pass the class object <sha256> etc. to the message digester; see Section 12.82 [Message digester framework], page 1017, and also see Section 12.45 [HMAC keyed-hashing], page 928.

rfc.sha1 [Module]
This is the old module that provided only SHA-1. It is kept as an alias of rfc.sha for the backward compatibility. New code should use rfc.sha.

<sha1> [Class]
<sha224> [Class]
<sha256> [Class]
<sha384> [Class]
<sha512> [Class]

{rfc.sha} An instance of these class keeps internal state of SHA digest algorithm. This class implements util.digest framework interface, so you can pass this class to message digest procedures such as digest-message-to (see Section 12.82 [Message digester framework], page 1017).

The following procedures are deprecated. Use generic message digester (see Section 12.82 [Message digester framework], page 1017) or HMAC keyed digester (see Section 12.45 [HMAC keyed-hashing], page 928).

sha1-digest [Function]
sha224-digest [Function]
sha256-digest [Function]
sha384-digest [Function]
sha512-digest [Function]

{rfc.sha} Deprecated. Reads data from the current input port until EOF, and returns its digest in an incomplete string.

sha1-digest-string string [Function]
sha224-digest-string string [Function]
sha256-digest-string string [Function]
sha384-digest-string string [Function]
sha512-digest-string string [Function]

{rfc.sha} Deprecated. Digest the data in string, and returns the result in an incomplete string.

12.54 rfc.tls - Transport layer security

rfc.tls [Module]
This module handles secure connection over TCP socket. This module is used by rfc.http to realize https connection (see Section 12.46 [HTTP client], page 930).
CA certificates

You need CA certificates to verify server certificates properly. Gauche doesn’t have its own CA certificates, and relies on the system’s certificate store by default. On Unix-based systems, we search several known locations: On popular Linux distributions we recommend you to install `ca-certificates` package (or similar one). On OSX, we recommend to install openssl via Homebrew. On Windows, we use system’s certificate store via Wincrypt API.

With the default configuration, Gauche checks the availability of the system’s certificate store at initialization and use one if available. You can explicitly give the path of CA certificate bundle, or disable it and provide individual certificate per connection.

If, for some reasons, you cannot install system-wide CA certificate bundle, you can also download Curl’s CA certificate bundle https://curl.haxx.se/ca/cacert.pem, and install it in Gauche installation directory. We have a convenience script. After installing Gauche, run the following command:

```
gosh tools/get-cacert.scm
```

You need curl installed on your system. If you’ve installed Gauche with root priviledge, you’ll be asked sudo password to install the CA bundle file.

If you decided to do this, make sure you run the above command occasionally to get updated CA certificate bundles, for certificates may expire or be revoked.

```
tls-ca-bundle-path :optional path
```

{rfc.tls} Holds CA certificate bundle to be used. The value can be either a string path to a CA bundle file, a symbol `system`, or `#f`.

If it is `system`, Gauche uses system’s default bundle. An error is signaled on connection if Gauche can’t find one.

If it is `#f`, CA certificate won’t be loaded automatically, and you have to manually load one using `tls-load-object`. (Note: This option is only valid with `<ax-tls>`. If you’re using `<mbed-tls>`, you need valid CA certificate bundle.)

With the default configuration, Gauche scans the system CA bundle when `rfc.tls` module is initialized, and if it finds one, `tls-ca-bundle-path` is set to `system`; otherwise, `tls-ca-bundle-path` is set to `#f`. So if you’re using with default configuration and you see its value is `system`, you can count on the system CA certificate bundle.

This default behavior may be altered if Gauche is configured with `--with-ca-bundle` option. You can execute `gauche-config --reconfigure` command to see if special `--with-ca-bundle` option is given.

Supported TLS subsystems

Gauche supports two TLS subsystems - one based on MbedTLS (https://tls.mbed.org/), and the other based on AxTLS (http://axtls.sourceforge.net/). Whether they’re included depends on the configuration options. By default, MbedTLS support is included if if the build platform has MbedTLS library installed. You can also include MbedTLS embedded in Gauche so that the runtime won’t need MbedTLS library installed on the system. See INSTALL.adoc in the source tree for the details.

Gauche chooses available TLS subsystem at runtime, so the user usually does not need to worry about the difference. If you’re building Gauche by yourself, we recommend to use the default configuration, on a system that has MbedTLS installed. That will give you the most flexible choice.

Whether the running Gauche has any of TLS support can be checked with a feature identifier `gauche.net.tls`. Availability of each individual subsystems can be checked with feature identifiers `gauche.net.tls.axtls` and `gauche.net.tls.mbedtls`, respectively. See Section 4.12 [Feature conditional], page 75, for more about feature identifiers.
<tls>
{rfc.tls} An abstract base class of TLS implementations.
</tls>

<mbed-tls>
{rfc.tls} A class that implements mbedTLS subsystem interface.
</mbed-tls>

<ax-tls>
{rfc.tls} A class that implements axTLS subsystem interface.
</ax-tls>

default-tls-class :optional class
{rfc.tls} Set/get the default TLS subsystem to be used. Without arguments, it return a
class (either <ax-tls> or <mbed-tls>) to be used. With one argument, which must be either
<ax-tls> or <mbed-tls>, changes the default and returns the previous value.

TLS interface

make-tls initargs ...
{rfc.tls} Creates and returns a new TLS instance of the class default-tls-class. The
returned TLS instance can be used for the tls procedures below.

make-tls
:server-name
Server name to be used for TLS Server Name Indication extension.

make-tls
:num-sessions
[AxTLS only] The number of sessions to be used for session caching. 0 indicates
no sessino caching.

make-tls
:options
[AxTLS only] Bitflags of options. Logical OR of the following numeric constants.
SSL_SERVER_VERIFY_LATER
(client only) Let handshake success even the server authentication fails.

make-tls
:options
SSL_CLIENT_AUTHENTICATION
(server only) Request client certificate to be authenticated.

make-tls
:options

tls-connect tls host port proto
{rfc.tls} Establishes TLS connection as a client. The first form is the recommended API.
The second form is only kept for the backward compatibility, may not work on the newer
MbedTLS, and will be removed in future.
The tls argument must be an unconnected TLS object. The host and port arguments are
strings to specify server’s hostname and port. Besides the common hostname, IP notation
(e.g. ”127.0.0.1” or ”[::1]”) are allowed in host. Numeric notation (e.g. ”443”) or service
name (e.g. ”https”) are allowed in port. The proto argument must be either one of the
symbols tcp or udp.

On success, it returns tls, which is in connected state. It can be used to read/write data
from/to the connected peer.

In the second, deprecated form, the socket argument must be a connected socket. The socket
is owned by tls object and should not be directly accessed after calling this procedure.

tls-accept tls socket
{rfc.tls} Deprecated. This is incompatible to the newer MbedTLS. We plan to provide
an alternative API, with which you can “bind” a tls object and then “accept” the client
connection, without touching the underlying sockets.

Establishes TLS connection as a server. The tls argument must be an unconnected TLS
object, and the socket argument must be a listening socket. On success, it returns tls, which
is in connected state. It can be used to read/write data from/to the connected peer.
tls-close tls
   {rfc.tls} Shuts down the underlying connection. The peer will be notified the connection is closed. Once tls is closed, it can no longer be used, but the resources won’t be collected until tls is GC-ed or tls-destroy is called. We recommend the user call tls-destroy explicitly.

tls-destroy tls
   {rfc.tls} Releases resources associated to tls.

tls-input-port tls

tls-output-port tls
   {rfc.tls} If tls is connected, it returns input and output port to communicate with the peer, respectively.
   If tls is not connected, #f is returned.

tls-load-object tls type filename :optional password
   {rfc.tls} This procedure is only meaningful for <axl-tls>. For <mbed-tls>, this procedure does nothing.
   Load private keys or certificates stored in a file specified by filenames. The type of object is specified by type argument with one of the following numeric constants. If the file requires a password, it should be given to password.

   SSL_OBJ_X509_CERT
   SSL_OBJ_X509_CACERT
   SSL_OBJ_RSA_KEY
   SSL_OBJ_PKCS8
   SSL_OBJ_PKCS12

12.55 rfc.uri - URI parsing and construction

rfc.uri
   [Module]
   Provides a set of procedures to parse and construct Uniform Resource Identifiers defined in RFC 2396 (https://www.ietf.org/rfc/rfc2396.txt), as well as Data URI scheme defined in RFC2397.

   First, let’s review the structure of URI briefly. The following graph shows how the URI is constructed:

   URI---scheme
      |   ---specific---authority---userinfo
      |       ---host
      |       ---port
      +---path
      +---query
      +---fragment

   Not all URIs have this full hierarchy. For example, mailto:admin@example.com has only scheme (mailto) and specific (admin@example.com) parts.
Most popular URI schemes, however, organize resources in a tree, so they adopt *authority* (which usually identifies the server) and the hierarchical *path*. In the URI `http://example.com:8080/search?q=key#results`, the authority part is `example.com:8080`, the path is `/search`, the query is `key` and the fragment is `results`. The userinfo can be provided before hostname, such as `anonymous` in `ftp://anonymous@example.com/pub/`.

We have procedures that decompose a URI into those parts, and that compose a URI from those parts.

**Parsing URI**

```scheme
(uri-ref uri 'parts)  \[Function\]
{rfc.uri} Extract specific part(s) from the given URI. You can fully decompose URI by
the procedures described below, but in actual applications, you often need only some of the
parts. This procedure comes handy for it.
The *parts* argument may be a symbol, or a list of symbols, to name the desired parts. The
recognized symbols are as follows.
scheme The scheme part, as string.
authority The authority part, as string. If URI doesn’t have the part, #f.
userinfo The userinfo part, as string. If URI doesn’t have the part, #f.
host The host part, as string. If URI doesn’t have the part, #f.
port The port part, as integer. If URI doesn’t have the part, #f.
path The path part, as string. If URI isn’t hierarchical, this returns the specific part.
query The query part, as string. If URI doesn’t have the part, #f.
fragment The fragment part, as string. If URI doesn’t have the part, #f.
scheme+authority The scheme and authority part.
host+port The host and port part.
userinfo+host+port The userinfo, host and port part.
path+query The path and query part.
path+query+fragment The path, query and fragment part.
```

(define uri "http://foo:bar@example.com:8080/search?q=word#results")

- (uri-ref uri 'scheme) ⇒ "http"
- (uri-ref uri 'authority) ⇒ "/foo:bar@example.com:8080/"
- (uri-ref uri 'userinfo) ⇒ "foo:bar"
- (uri-ref uri 'host) ⇒ "example.com"
- (uri-ref uri 'port) ⇒ 8080
- (uri-ref uri 'path) ⇒ "/search"
- (uri-ref uri 'query) ⇒ "q=word"
- (uri-ref uri 'fragment) ⇒ "results"
- (uri-ref uri 'scheme+authority) ⇒ "http://foo:bar@example.com:8080/"
You can extract multiple parts at once by specifying a list of parts. A list of parts is returned.

```
(quote (uri-ref uri 'host+port)) ⇒ "example.com:8080"
(quote (uri-ref uri 'userinfo+host+port)) ⇒ "foo:bar@example.com:8080"
(quote (uri-ref uri 'path+query)) ⇒ "/search?q=word"
(quote (uri-ref uri 'path+query+fragment)) ⇒ "/search?q=word#results"
```

The following procedures are finer grained and break up uris with different stages.

**uri-scheme&specific** takes a URI *uri*, and returns two values, its scheme part and its scheme-specific part. If *uri* doesn’t have a scheme part, #f is returned for it.

```
(quote (uri-scheme&specific "mailto:sclaus@north.pole")) ⇒ "mailto" and "sclaus@north.pole"
(quote (uri-scheme&specific "/icons/new.gif")) ⇒ #f and "/icons/new.gif"
```

If the URI scheme uses hierarchical notation, i.e. “//authority/path?query#fragment”, you can pass the scheme-specific part to **uri-decompose-hierarchical** and it returns four values, authority, path, query and fragment.

```
(quote (uri-decompose-hierarchical "///www.foo.com/about/company.html")) ⇒ "www.foo.com", "/about/company.html", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/search?key=%3Fhelp")) ⇒ "www.localhost", "/search", "key=%3Fhelp" and #f
(quote (uri-decompose-hierarchical "///www.localhost/search2?param=42")) ⇒ "www.localhost", "/search2", "param=42" and #f
(quote (uri-decompose-hierarchical "///zzz.org/search?key=%3Fhelp")) ⇒ "zzz.org", "/search", "key=%3Fhelp" and #f
(quote (uri-decompose-hierarchical "///www.localhost/search")) ⇒ "www.localhost", "/search", and #f
(quote (uri-decompose-hierarchical "///zzz.org/search?key=xyz&lang=en")) ⇒ "zzz.org", "/search", "key=xyz&lang=en" and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
(quote (uri-decompose-hierarchical "///www.localhost/")) ⇒ "www.localhost", "/", #f and #f
```

Furthermore, you can parse **authority** part of the hierarchical URI by **uri-decompose-authority**. It returns userinfo, host and port.

```
(quote (uri-decompose-authority "//www.localhost:8080")) ⇒ "www.localhost", "/", 8080, #f and #f
```

⇒ #f, "yyy.jp" and "8080"
(uri-decompose-authority ":[::1]:8080") ; (IPv6 host address)
⇒ #f, ":1" and "8080"
( uri-decompose-authority "mylogin@yyy.jp")
⇒ "mylogin", "yyy.jp" and #f

uri-decompose-data uri {rfc.uri} Parse a Data URI string uri. You can either pass the entire uri including data: scheme part, or just the specific part. If the passed uri is invalid as a data uri, an error is signalled.

Returns two values: parsed content type and the decoded data. The data is a string if the content type is text/*, and a u8vector otherwise.

The content-type is parsed by mime-parse-content-type (see Section 12.51 [MIME message handling], page 939). The result format is a list as follows:

((type subtype (attribute . value) ...)).

Here are a couple of examples:

(.uri-decompose-data
 "data:text/plain;charset=utf-8;base64,KGhlbGxvIHdvcmxkKQ==")
⇒ ("text" "plain" ("charset" . "utf-8")) and "(hello world)"

( uri-decompose-data
 "data:application/octet-stream;base64,AAECAw==")
⇒ ("application" "octet-stream") and #u8(0 1 2 3)

Constructing URI

uri-compose :key scheme userinfo host port authority path path* query fragment specific [Function]
{rfc.uri} Compose a URI from given components. There can be various combinations of components to create a valid URI—the following diagram shows the possible ‘paths’ of combinations:

/-----------------specific-------------------
|       |       |
scheme-+------authority-----+-+-------path*---------+-
| | | |
\-userinfo-host-port/- \-path-query-fragment-/ 

If #f is given to a keyword argument, it is equivalent to the absence of that keyword argument. It is particularly useful to pass the results of parsed uri.

If a component contains a character that is not appropriate for that component, it must be properly escaped before being passed to uri-compose.

Some examples:

( uri-compose :scheme "http" :host "foo.com" :port 80
 :path "/index.html" :fragment "top")
⇒ "http://foo.com:80/index.html#top"

( uri-compose :scheme "http" :host "foo.net"
 :path* "/cgi-bin/query.cgi?keyword=foo")
⇒ "http://foo.net/cgi-bin/query.cgi?keyword=foo"

( uri-compose :scheme "mailto" :specific "a@foo.org")
(receive (authority path query fragment)
  (uri-decompose-hierarchical "//foo.jp/index.html#whatsnew")
  (uri-compose :authority authority :path path
    :query query :fragment fragment))
⇒ "//foo.jp/index.html#whatsnew"

uri-merge base-uri relative-uri relative-uri2 ... [Function]
{rfc.uri} Arguments are strings representing full or part of URIs. This procedure resolves
relative-uri in relative to base-uri, as defined in RFC3986 Section 5.2. “Relative Resolution”.
If more relative-uri2s are given, first relative-uri is merged to base-uri, then the next argument
is merged to the resulting uri, and so on.

((uri-merge "http://example.com/foo/index.html" "a/b/c")
⇒ "http://example.com/foo/a/b/c"

((uri-merge "http://example.com/foo/search?q=abc" "../about#me")
⇒ "http://example.com/about#me"

((uri-merge "http://example.com/foo" "http://example.net/bar")
⇒ "http://example.net/bar"

((uri-merge "http://example.com/foo/" "q" "?xyz")
⇒ "http://example.com/foo?q?xyz"

uri-compose-data data :key content-type encoding [Function]
{rfc.uri} Creates a Data URI of the given data, with specified content-type and transfer
encoding. Returns a string.
The data argument must be a string or a u8vector.
The content-type argument can be #f (default), a string that represents a content type (e.g.
"text/plain; charset=utf-8"), or a list form of parsed content type (e.g. ("application" 
"octet-stream"). If it is #f, text/plain with the gauche's native character encoding is
used when data is a complete string, and application/octet-stream is used otherwise.
The encoding argument can be either #f (default), or a symbol uri or base64. This is for
transfer encoding, not character encoding. If it is #f, URI encoding is used for text data and
base64 encoding is used for binary data.

((uri-compose-data "(hello world)")
⇒ "data:text/plain;charset=utf-8,%28hello%20world%29"

(uri-compose-data "(hello world)" :encoding 'base64)
⇒ "data:text/plain;charset=utf-8;base64,KGhlbGxvIHdvcmxkKQ=="

(uri-compose-data '#u8(0 1 2 3))
⇒ "data:application/octet-stream;base64,AAECAw=="

URI Encoding and decoding

uri-decode :key :cgi-decode [Function]
uri-decode-string string :key :cgi-decode :encoding [Function]
{rfc.uri} Decodes “URI encoding”, i.e. %-escapes. uri-decode takes input from the current
input port, and writes decoded result to the current output port. uri-decode-string takes
input from string and returns decoded string.
If **cgi-decode** is true, also replaces + to a space character.

To **uri-decode-string** you can provide the external character encoding by the **encoding** keyword argument. When it is given, the decoded octet sequence is assumed to be in the specified encoding and converted to the Gauche’s internal character encoding.

```scheme
[Function]
uri-encode :key :noescape
[Function]
uri-encode-string string :key :noescape :encoding
```

{rfc.uri} Encodes unsafe characters by %-escape. **uri-encode** takes input from the current input port and writes the result to the current output port. **uri-encode-string** takes input from **string** and returns the encoded string.

By default, characters that are not specified “unreserved” in RFC3986 are escaped. You can pass different character set to **noescape** argument to keep from being encoded. For example, the older RFC2396 has several more “unreserved” characters, and passing ***rfc2396-unreserved-char-set*** (see below) prevents those characters from being escaped.

The multibyte characters are encoded as the octet stream of Gauche’s native multibyte representation by default. However, you can pass the **encoding** keyword argument to **uri-encode-string**, to convert **string** to the specified character encoding.

```scheme
*rfc2396-unreserved-char-set*
*rfc3986-unreserved-char-set*
```

{rfc.uri} These constants are bound to character sets that represents “unreserved” characters defined in RFC2396 and RFC3986, respectively. (See Section 6.10 [Character sets], page 165, and Section 10.3.6 [R7RS character sets], page 614, for operations on character sets).

### 12.56 rfc.uuid - UUID

**rfc.uuid**

This module implements UUID defined in RFC4122.

It provides generators of UUID version 1 and 4, and writer/parser of the string representation of UUIDs.

```scheme
<uuid>
```

{rfc.uuid} Class of UUID instances. UUID instances are immutable.

```scheme
uuid-value uuid
```

{rfc.uuid} Returns the raw value of **uuid** as 16-element u8vector. You shouldn’t mutate the returned u8vector.

```scheme
uuid-version uuid
```

{rfc.uuid} Returns the version number of **uuid**.

```scheme
uuid-comparator
```

{rfc.uuid} A comparator to compare and hash uuids. See Section 6.2.4 [Basic comparators], page 116.

Note: Equality of uuids can be tested with **equal?**.

```scheme
uuid-random-source
```

{rfc.uuid} We use PRNG to generate UUIDs. By default, we internally creates a random source and randomize it. You can alter the random source using **parameterize**. The value must be a SRFI-27 random source (see Section 11.8 [Sources of random bits], page 706).
**uuid-random-source-set!**  
*random-source*  
{rfc.uuid} Deprecated. Use `uuid-random-source` parameter to customize random source to be used in uuid generation.

**uuid1 :optional node-id**  
{rfc.uuid} Generates a uuid with version 1 algorithm (timestamp and node id based). The optional `node-id` argument must be 48bit exact integer specifing the node ID (IEEE802 MAC address of the machine). If it is omitted, we generate a process-global random node ID (with the multicast bit set to 1, so that it won’t conflict with existing MAC address).

**uuid4**  
{rfc.uuid} Generates a uuid with version 4 algorithm (random numbers).

**nil-uuid**  
{rfc.uuid} Returns a nil-UUID (all bits zero).

**parse-uuid string :key if-invalid**  
{rfc.uuid} Takes a string representation of UUID, parses it and returns an uuid instance. If the string isn’t a valid UUID representation, an error is thrown if the `if-invalid` keyword argument is omitted or :error, and #f is returned if `if-invalid` is #f. Other than `XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX` format, it recognizes the one with `urn:uuid:` prefix, the one enclosed by curly braces, and the one without hyphens.

**write-uuid uuid :optional port**  
{rfc.uuid} Writes out a string representation of `uuid`, in `XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX` format, to the given port. If the port is omitted, current output port is used.

**uuid->string uuid**  
{rfc.uuid} Returns a string representation of `uuid`, in `XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX` format.

### 12.57 rfc.zlib - zlib compression library

**rfc.zlib**  
This module provides bindings to zlib compression library. Most features of zlib can be used through this module.

Zlib supports reading and writing of Zlib compressed data format (RFC1950), DEFLATE compressed data format (RFC1951), and GZIP file format (RFC1052). It also provides procedures to calculate CRC32 and Adler32 checksums.

Compression and decompression are done through specialized ports. There are number of parameters to fine-tune compression; refer to zlib documentation for the details.

**Condition types**  
The following condition types are defined to represent errors during processing by zlib.

**<zlib-error>**  
{rfc.zlib} Subclass of `<error>` and superclass of the following condition types. This class is an abstract class to catch any of the zlib-specific errors. Zlib-specific errors raised by procedures in `rfc.zlib` are always an instance (or a compound condition including) one of the following specific classes.
<zlib-need-dict-error> [Condition Type]
URRE <zlib-stream-error> [Condition Type]
<zlib-data-error> [Condition Type]
<zlib-memory-error> [Condition Type]
<zlib-version-error> [Condition Type]

{rfc.zlib} Subclasses of <zlib-error>. Those condition type correspond to zlib's Z_NEED_DICT_ERROR, Z_STREAM_ERROR, Z_DATA_ERROR, Z_MEMORY_ERROR, and Z_VERSION_ERROR errors.

When an error occurs during reading data, a compound condition of a subclass of <zlib-error> and <io-read-error> is raised. When an error occurs without I/O, a simple condition of a subclass of <zlib-error> is raised. Errors unrelated to zlib, such as invalid argument error, would be a simple <error> condition.

**Compression/decompression ports**

<deflating-port> [Class]
<inflating-port> [Class]

{rfc.zlib} Compression and decompression functions are provided via ports. A deflating port is an output port that compresses the output data. An inflating port is an input that reads compressed data and decompress it.

When an inflating port encounters a corrupted compressed data, a compound condition of <io-read-error> and <zlib-data-error> is raised during read operation.

**open-deflating-port** drain :key compression-level buffer-size

{rfc.zlib} Creates and returns an instance of <deflating-port>, an output port that compresses the output data and sends the compressed data to another output port drain. This combines the functionality of zlib's deflateInit2() and deflateSetDictionary().

You can specify an exact integer between 1 and 9 (inclusive) to compression-level. Larger integer means larger compression ratio. When omitted, a default compression level is used, which is usually 6.

The following constants are defined to specify compression-level conveniently:

Z_NO_COMPRESSION  [Constant]
Z_BEST_SPEED  [Constant]
Z_BEST_COMPRESSION  [Constant]
Z_DEFAULT_COMPRESSION  [Constant]

{rfc.zlib}

The buffer-size argument specifies the buffer size of the port in bytes. The default is 4096.

The window-bits argument specifies the size of the window in exact integer. Typically the value should be between 8 and 15, inclusive, and it specifies the base two logarithm of the window size used in compression. Larger number yields better compression ratio, but more memory usage. The default value is 15.

There are a couple of special modes specifiable by window-bits. When an integer between -8 and -15 is given to window-bits, the port produces a raw deflated data, that lacks zlib header and trailer. In this case, Adler32 checksum isn’t calculated. The actual window size is determined by the absolute value of window-bits.

When window-bits is between 24 and 31, the port uses GZIP encoding; that is, instead of zlib wrapper, the compressed data is enveloped by simple gzip header and trailer. The gzip header added by this case doesn’t have filenames, comments, header CRC and other data, and have zero modified time, and 255 (unknown) in the OS field. The zstream-adler32 procedure
will return CRC32 checksum instead of Adler32. The actual window size is determined by window-bits-16.

The memory-level argument specifies how much memory should be allocated to keep the internal state during compression. 1 means smallest memory, which causes slow and less compression. 9 means fastest and best compression with largest amount of memory. The default value is 8.

To fine tune compression algorithm, you can use the strategy argument. The following constants are defined as the valid value as strategy:

- **Z_DEFAULT_STRATEGY**
  {rfc.zlib} The default strategy, suitable for most ordinary data.

- **Z_FILTERED**
  {rfc.zlib} Suitable for data generated by filters. Filtered data consists mostly of small values with a random distribution, and this makes the compression algorithm to use more huffman encoding and less string match.

- **Z_HUFFMAN_ONLY**
  {rfc.zlib} Force huffman encoding only (no string match).

- **Z_RLE**
  {rfc.zlib} Limit match distance to 1 (that is, to force run-length encoding). It is as fast as Z_HUFFMAN_ONLY and gives better compression for png image data.

- **Z_FIXED**
  {rfc.zlib} Prohibits dynamic huffman encoding. It allows a simple decoder for special applications.

The choice of strategy only affects compression ratio and speed. Any choice produces correct and decompressable data.

You can give an initial dictionary to the dictionary argument to be used in compression. The compressor and decompressor must use exactly the same dictionary. See the zlib documentation for the details.

By default, a deflating port leaves drain open after all conversion is done, i.e. the deflating port itself is closed. If you don’t want to bother closing drain, give a true value to the owner? argument; then drain is closed after the deflating port is closed and all data is written out.

Note: You must close a deflating port explicitly, or the compressed data can be chopped prematurely. When you leave a deflating port open to be GCed, the finalizer will close it; however, the order in which finalizers are called is undeterministic, and it is possible that the drain port is closed before the deflating port is closed. In such cases, the deflating port’s attempt to flush the buffered data and trailer will fail.

The meaning of buffer-size and owner are the same as open-deflating-port.

The meaning of window-bits is almost the same, except that if a value increased by 32 is given, the inflating port automatically detects whether the source stream is zlib or gzip by its header.
That is, you can specify between 8 to 15 to read zlib, 24 to 31 to read gzip, or 40 to 47 to use automatic detection.

The window bits must be equal to or greater than the window bits used to compress the source, or a `<zlib-data-error>` condition is thrown. If you don’t know the compression parameters of the input (which is most likely the case), you need to specify the maximum value, i.e. 15 for zlib, 31 for gzip, or 47 to autodetect.

If the input data is compressed with specified dictionary, the same dictionary must be given to the `dictionary` argument. Otherwise, a compound condition of `<io-read-error>` and `<zlib-need-dict-error>` will be raised.

**Operations on inflating/deflating ports**

```lisp
zstream-total-in xflating-port [Function]
zstream-total-out xflating-port [Function]
zstream-adler32 xflating-port [Function]
zstream-data-type xflating-port [Function]
  {rfc.zlib} The `xflating-port` argument must be either inflating and deflating port, or an error is raised.

Returns the value of `total_in`, `total_out`, `adler32`, and `data_type` fields of the `z_stream` structure associated to the given inflating or deflating port, respectively.

The value of `data_type` can be one of the following constants:

- `Z_BINARY`
- `Z_TEXT`
- `Z_ASCII`
- `Z_UNKNOWN`
  {rfc.zlib}

zstream-params-set! deflating-port :key compression-level strategy [Function]
  {rfc.zlib} Changes compression level and/or strategy during compressing.

zstream-dictionary-adler32 deflating-port [Function]
  {rfc.zlib} When a dictionary is given to `open-deflating-port`, the dictionary’s adler32 checksum is calculated. This procedure returns the checksum. If no dictionary has been given, this procedure returns `#f`.

deflating-port-full-flush deflating-port [Function]
  {rfc.zlib} Flush the data buffered in the `deflating-port`, and resets compression state. The decompression routine can skip the data to the full-flush point by `inflate-sync`.

inflate-sync inflating-port [Function]
  {rfc.zlib} Skip the (possibly corrupted) compressed data up to the next full-flush point marked by `deflating-port-full-flush`. You may want to use this procedure when you get `<zlib-data-error>`. Returns the number of bytes skipped when the next full-flush point is found, or `#f` when the input reaches EOF before finding the next point.

**Miscellaneous API**

zlib-version [Function]
  {rfc.zlib} Returns Zlib’s version in string.

deflate-string string options ... [Function]
  {rfc.zlib} Compresses the given string and returns zlib-compressed data in a string. All optional arguments are passed to `open-deflating-port` as they are.
inflation-string string options . . .
{rfc.zlib} Takes zlib-compressed data in string, and returns decompressed data in a string.
All optional arguments are passed to open-inflating-port as they are.

gzip-encode-string string options . . .
gzip-decode-string string options . . .
{rfc.zlib} Like deflate-string and inflate-string, but uses the gzip format instead.
It is same as giving more than 15 to the window-bits argument of deflate-string and
inflate-string.

crc32 string :optional checksum
{rfc.zlib} Returns CRC32 checksum of string. If optional checksum is given, the returned
checksum is an update of checksum by string.
adler32 string :optional checksum
{rfc.zlib} Returns Adler32 checksum of string. If optional checksum is given, the returned
checksum is an update of checksum by string.
Calculating Adler32 is faster than CRC32, but it is known to produce uneven distribution
of hash values for small input. See RFC3309 for the detailed description. If it matters, use
CRC32 instead.

12.58 slib - SLIB interface

slib
This module is the interface to the Aubrey Jaffer’s SLIB. To use SLIB, say (use slib). SLIB
itself is not included in Gauche distribution. If you don’t have it on your system, get it from
By default, the SLIB installation is searched from the directory specified at the Gauche con-
figuration. If SLIB isn’t there, an error is signaled. In that case, you can set the environment
variable SCHEME_LIBRARY_PATH to point to the SLIB installation path.
This module redefines require, shadowing the Gauche’s original require. If it gets a symbol
as an argument, it works as SLIB’s require, while if it gets a string, it works as Gauche’s
require. The same applies to provide and provided?.
All SLIB symbol bindings, loaded by require, stay in the module slib.
NB: SLIB probes available srfis during initialization, and by the way it does so, all available
srfis are loaded, regardless of whether you require it or not. This may introduce unexpected
side effects; for example, Gauche’s built-in regexp-replace is shadowed by srfi.115’s one
(see Section 10.3.19 [R7RS regular expressions], page 640), which has slightly different API.

12.59 sxml.ssax - Functional XML parser

sxml.ssax
sxml.* modules are the adaptation of Oleg Kiselyov’s SXML framework (http://okmij.
org/ftp/Scheme/xml.html), which is based on S-expression representation of XML struc-
ture.
SSAX is a parser part of SXML framework. This is a quote from SSAX webpage:
A SSAX functional XML parsing framework consists of a DOM/SXML parser,
a SAX parser, and a supporting library of lexing and parsing procedures. The
procedures in the package can be used separately to tokenize or parse various pieces of XML documents. The framework supports XML Namespaces, character, internal and external parsed entities, attribute value normalization, processing instructions and CDATA sections. The package includes a semi-validating SXML parser: a DOM-mode parser that is an instantiation of a SAX parser (called SSAX).

The current version is based on the SSAX CVS version newer than the last ’official’ release of SXML toolset (4.9), and SXML-gauche-0.9 package which was based on SXML-4.9. There is an important change from that release. Now the API uses lowercase letter suffix \texttt{ssax} instead of uppercase \texttt{SSAX}:—the difference matters since Gauche is case sensitive by default. Alias names are defined for backward compatibility, but the use of uppercase suffixed names are deprecated.

I derived the content of this part of the manual from SSAX source code, just by converting its comments into texinfo format. The original text is by Oleg Kiselyov. Shiro Kawai should be responsible for any typographical error or formatting error introduced by conversion.

The manual entries are ordered in "bottom-up" way, beginning from the lower-level constructs towards the high-level utilities. If you just want to parse XML document and obtain SXML, check out \texttt{ssax:xml->sxml} in Section 12.59.4 [SSAX Highest-level parsers - XML to SXML], page 967.

12.59.1 SSAX data types

\texttt{TAG-KIND}

a symbol \texttt{’START}, \texttt{’END}, \texttt{’PI}, \texttt{’DECL}, \texttt{’COMMENT}, \texttt{’CDSECT} or \texttt{’ENTITY-REF} that identifies a markup token.

\texttt{UNRES-NAME}

a name (called \texttt{GI} in the XML Recommendation) as given in an xml document for a markup token: start-tag, \texttt{PI} target, attribute name. If a \texttt{GI} is an \texttt{NCName}, \texttt{UNRES-NAME} is this \texttt{NCName} converted into a Scheme symbol. If a \texttt{GI} is a \texttt{QName}, \texttt{UNRES-NAME} is a pair of symbols: \texttt{(PREFIX . LOCALPART)}

\texttt{RES-NAME}

An expanded name, a resolved version of an \texttt{UNRES-NAME}. For an element or an attribute name with a non-empty namespace URI, \texttt{RES-NAME} is a pair of symbols, \texttt{(URI-SYMB . LOCALPART)}. Otherwise, it’s a single symbol.

\texttt{ELEM-CONTENT-MODEL}

A symbol:

\texttt{ANY} anything goes, expect an END tag.

\texttt{EMPTY-TAG} no content, and no END-tag is coming.

\texttt{EMPTY} no content, expect the END-tag as the next token.

\texttt{PCDATA} expect character data only, and no children elements.

\texttt{MIXED}

\texttt{ELEM-CONTENT}

\texttt{URI-SYMB}

A symbol representing a namespace URI – or other symbol chosen by the user to represent URI. In the former case, \texttt{URI-SYMB} is created by \texttt{%-} quoting of bad URI characters and converting the resulting string into a symbol.

\texttt{NAMESPACES}

A list representing namespaces in effect. An element of the list has one of the following forms:
(prefix uri-symb . uri-symb)
   or,

(promise user-prefix . uri-symb)
   user-prefix is a symbol chosen by the user to represent the URI.

(#f user-prefix . uri-symb)
   Specification of the user-chosen prefix and a uri-symbol.

(*DEFAULT* user-prefix . uri-symb)
   Declaration of the default namespace

(*DEFAULT* #f . #f)
   Un-declaration of the default namespace. This notation represents over-
   riding of the previous declaration

A NAMESPACE list may contain several elements for the same PREFIX. The one closest to the beginning of the list takes effect.

**ATTLIST**
   An ordered collection of (NAME . VALUE) pairs, where NAME is a RES-NAME
   or an UNRES-NAME. The collection is an ADT.

**STR-HANDLER**
   A procedure of three arguments: (string1 string2 seed) returning a new seed.
   The procedure is supposed to handle a chunk of character data string1 followed by
   a chunk of character data string2. string2 is a short string, often "\n" and even ""

**ENTITIES**
   An assoc list of pairs:
      (named-entity-name . named-entity-body)
   where named-entity-name is a symbol under which the entity was declared, named-
   entity-body is either a string, or (for an external entity) a thunk that will return
   an input port (from which the entity can be read). named-entity-body may also be
   #f. This is an indication that a named-entity-name is currently being expanded. A
   reference to this named-entity-name will be an error: violation of the WFC nonre-
   cursion.

**XML-TOKEN**
   A record with two slots, kind and token. This record represents a markup, which
   is, according to the XML Recommendation, "takes the form of start-tags, end-
   tags, empty-element tags, entity references, character references, comments, CDATA
   section delimiters, document type declarations, and processing instructions."

   kind       a TAG-KIND
   head       an UNRES-NAME. For xml-tokens of kinds 'COMMENT and 'CDSECT, the
               head is #f

   For example,
               <P>   => kind='START, head='P
               </P> => kind='END, head='P
               <BR/> => kind='EMPTY-EL, head='BR
               <!DOCTYPE OMF ...> => kind='DECL, head='DOCTYPE
               <?xml version="1.0"?> => kind='PI, head='xml
               &my-ent; => kind = 'ENTITY-REF, head='my-ent

   Character references are not represented by xml-tokens as these references are trans-
   parently resolved into the corresponding characters.
**XML-DECL**

A record with three slots, `elems`, `entities`, and `notations`.

The record represents a datatype of an XML document: the list of declared elements and their attributes, declared notations, list of replacement strings or loading procedures for parsed general entities, etc. Normally an xml-decl record is created from a DTD or an XML Schema, although it can be created and filled in in many other ways (e.g., loaded from a file).

- `elems`: an (assoc) list of decl-elem or `#f`. The latter instructs the parser to do no validation of elements and attributes.
- `decl-elem`: declaration of one element: `(elem-name elem-content decl-attrs);` `elem-name` is an UNRES-NAME for the element. `elem-content` is an ELEM-CONTENT-MODEL. `decl-attrs` is an ATTLIST, of `(attr-name . value)` associations. This element can declare a user procedure to handle parsing of an element (e.g., to do a custom validation, or to build a hash of IDs as they’re encountered).
- `decl-attr`: an element of an ATTLIST, declaration of one attribute `(attr-name content-type use-type default-value);` `attr-name` is an UNRES-NAME for the declared attribute; `content-type` is a symbol: `CDATA`, `NMTOKEN`, `NMTOKENS`, ...; or a list of strings for the enumerated type. `use-type` is a symbol: `REQUIRED`, `IMPLIED`, `FIXED` default-value is a string for the default value, or `#f` if not given.

```scheme
(make-empty-attlist) [Function]
(attlist-add attlist name-value) [Function]
(attlist-null?) [Function]
(attlist-remove-top attlist) [Function]
(attlist->alist attlist) [Function]
(attlist-fold) [Function]
```

{sxml.ssax} Utility procedures to deal with attribute list, which keeps name-value association.

```scheme
(make-xml-token kind head) [Function]
(xml-token? token) [Function]
```

{sxml.ssax} A constructor and a predicate for a XML-TOKEN record.

```scheme
(xml-token-kind token) [Macro]
(xml-token-head token) [Macro]
```

{sxml.ssax} Accessor macros of a XML-TOKEN record.

### 12.59.2 SSAX low-level parsing code

They deal with primitive lexical units (Names, whitespaces, tags) and with pieces of more generic productions. Most of these parsers must be called in appropriate context. For example, `ssax:complete-start-tag` must be called only when the start-tag has been detected and its GI has been read.

```scheme
(ssax:skip-S port) [Function]
```

{sxml.ssax} Skip the S (whitespace) production as defined by

[3] S ::= (#x20 | #x9 | #xD | #xA)

The procedure returns the first not-whitespace character it encounters while scanning the `port`. This character is left on the input stream.

```scheme
(ssax:ncname-starting-char? a-char) [Function]
```

{sxml.ssax} Check to see if a-char may start a NCName.
ssax:read-NCName port
{sxml.ssax} Read a NCName starting from the current position in the port and return it as a symbol.

ssax:read-QName port
{sxml.ssax} Read a (namespace-) Qualified Name, QName, from the current position in the port.
From REC-xml-names:
[6] QName ::= (Prefix ':')? LocalPart
[7] Prefix ::= NCName
[8] LocalPart ::= NCName

Return: an UNRES-NAME.

ssax:Prefix-XML
{sxml.ssax} The prefix of the pre-defined XML namespace, i.e. 'xml.'

ssax:read-markup-token port
{sxml.ssax} This procedure starts parsing of a markup token. The current position in the stream must be #<. This procedure scans enough of the input stream to figure out what kind of a markup token it is seeing. The procedure returns an xml-token structure describing the token. Note, generally reading of the current markup is not finished! In particular, no attributes of the start-tag token are scanned.
Here's a detailed break out of the return values and the position in the port when that particular value is returned:

PI-token only PI-target is read. To finish the Processing Instruction and disregard it, call ssax:skip-pi. ssax:read-attributes may be useful as well (for PIs whose content is attribute-value pairs)

END-token The end tag is read completely; the current position is right after the terminating #\> character.

COMMENT is read and skipped completely. The current position is right after "--" that terminates the comment.

CDSECT The current position is right after "<![CDATA[". Use ssax:read-cdata-body to read the rest.

DECL We have read the keyword (the one that follows "<!") identifying this declaration markup. The current position is after the keyword (usually a whitespace character)

START-token We have read the keyword (GI) of this start tag. No attributes are scanned yet. We don’t know if this tag has an empty content either. Use ssax:complete-start-tag to finish parsing of the token.

ssax:skip-pi port
{sxml.ssax} The current position is inside a PI. Skip till the rest of the PI.

ssax:read-pi-body-as-string port
{sxml.ssax} The current position is right after reading the PITarget. We read the body of PI and return it as a string. The port will point to the character right after '?' combination that terminates PI.

[16] PI ::= ’<?’ PITarget (S (Char* - (Char* ’?’ Char*)))?’?’
ssax:skip-internal-dtd port  [Function]
{sxml.ssax} The current pos in the port is inside an internal DTD subset (e.g., after reading 
that begins an internal DTD subset) Skip until the "\]>" combination that terminates this DTD

ssax:read-cdata-body port str-handler seed  [Function]
{sxml.ssax} This procedure must be called after we have read a string "<![CDATA[" that be-
gins a CDATA section. The current position must be the first position of the CDATA body. This
function reads lines of the CDATA body and passes them to a STR-HANDLER, a character
data consumer.

The str-handler is a STR-HANDLER, a procedure string1 string2 seed. The first string1
argument to STR-HANDLER never contains a newline. The second string2 argument of-
ten will. On the first invocation of the STR-HANDLER, the seed is the one passed to
ssax:read-cdata-body as the third argument. The result of this first invocation will be
passed as the seed argument to the second invocation of the line consumer, and so on. The
result of the last invocation of the STR-HANDLER is returned by the ssax:read-cdata-
body. Note a similarity to the fundamental 'fold' iterator.

Within a CDATA section all characters are taken at their face value, with only three exceptions:

- CR, LF, and CRLF are treated as line delimiters, and passed as a single 
newline to the
STR-HANDLER.
- "\]]>" combination is the end of the CDATA section.
- &gt; is treated as an embedded ">
character. Note, &lt; and &amp; are not specially
recognized (and are not expanded)!

ssax:read-char-ref port  [Function]
{sxml.ssax}

[66] CharRef ::= '&#' [0-9]+ ';'
| '&#x' [0-9a-fA-F]+ ';'

This procedure must be called after we we have read "&#" that introduces a char reference.
The procedure reads this reference and returns the corresponding char. The current po-
position in port will be after ";" that terminates the char reference. Faults detected: WFC:
XML-Spec.html#wf-Legalchar.

According to Section "4.1 Character and Entity References" of the XML Recommendation:

"[Definition: A character reference refers to a specific character in the ISO/IEC
10646 character set, for example one not directly accessible from available input
devices.]

Therefore, we use a ucscode->char function to convert a character code into the character
– regardless of the current character encoding of the input stream.

ssax:handle-parsed-entity port name entities content-handler  [Function]
{sxml.ssax} Expand and handle a parsed-entity reference

- port - a PORT
- name - the name of the parsed entity to expand, a symbol.
- entities - see ENTITIES
- content-handler - procedure port entities seed that is supposed to return a seed.
- str-handler - a STR-HANDLER. It is called if the entity in question turns out to be a
pre-declared entity
The result is the one returned by `content-handler` or `str-handler`.

Faults detected:

\[\text{WFC: XML-Spec.html#wf-entdeclared}\]
\[\text{WFC: XML-Spec.html#norecursion}\]

**ssax:read-attributes** \(\text{port entities}\)  
\{sxml.ssax\} This procedure reads and parses a production \(\text{Attribute}^*\)

\[[41] \text{Attribute} ::= \text{Name Eq AttValue} \]
\[[10] \text{AttValue} ::= \text{'' (\[^<&"] | \text{Reference}\)* ''} \]
\[[25] \text{Eq} ::= \text{S? '=' S?} \]

The procedure returns an `ATTLIST`, of `Name` (as `UNRES-NAME`), `Value` (as string) pairs. The current character on the `port` is a non-whitespace character that is not an ncnamesetarting character.

Note the following rules to keep in mind when reading an `AttValue` "Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize it as follows:

- a character reference is processed by appending the referenced character to the attribute value
- an entity reference is processed by recursively processing the replacement text of the entity [see ENTITIES] [named entities amp lt gt quot apos are assumed pre-declared]
- a whitespace character (#x20, #xD, #xA, #x9) is processed by appending #x20 to the normalized value, except that only a single #x20 is appended for a "#xD#xA" sequence that is part of an external parsed entity or the literal entity value of an internal parsed entity
- other characters are processed by appending them to the normalized value "

Faults detected:

\[\text{WFC: XML-Spec.html#CleanAttrVals}\]
\[\text{WFC: XML-Spec.html#uniqattspec}\]

**ssax:resolve-name** \(\text{port unres-name namespaces apply-default-ns?}\)  
\{sxml.ssax\} Convert an `unres-name` to a `res-name` given the appropriate `namespaces` declarations. The last parameter `apply-default-ns?` determines if the default namespace applies (for instance, it does not for attribute names)

Per REC-xml-names/#nsc-NSDeclared, "xml" prefix is considered pre-declared and bound to the namespace name "http://www.w3.org/XML/1998/namespace".

This procedure tests for the namespace constraints: http://www.w3.org/TR/REC-xml-names/#nsc-NSDeclared.

**ssax:uri-string->symbol** \(\text{uri-str}\)  
\{sxml.ssax\} Convert a `uri-str` to an appropriate symbol.

**ssax:complete-start-tag** \(\text{tag port elems entities namespaces}\)  
\{sxml.ssax\} This procedure is to complete parsing of a start-tag markup. The procedure must be called after the start tag token has been read. `Tag` is an `UNRES-NAME`. `Elem` is an instance of `xml-decl::elems`; it can be `#f` to tell the function to do no validation of elements and their attributes.

This procedure returns several values:

\(\text{elem-gi} \quad a \text{ RES-NAME}\).
attributes  element’s attributes, an ATTLIST of (res-name . string) pairs. The list does not include xmlns attributes.

namespaces  the input list of namespaces amended with namespace (re-)declarations contained within the start-tag under parsing ELEM-CONTENT-MODEL.

On exit, the current position in port will be the first character after #\> that terminates the start-tag markup.

Faults detected:
VC: XML-Spec.html#enum
VC: XML-Spec.html#RequiredAttr
VC: XML-Spec.html#FixedAttr
VC: XML-Spec.html#ValueType
WFC: XML-Spec.html#uniqattspec (after namespaces prefixes are resolved)
VC: XML-Spec.html#elementvalid
WFC: REC-xml-names/#dt-NSName

Note, although XML Recommendation does not explicitly say it, xmlns and xmlns: attributes don’t have to be declared (although they can be declared, to specify their default value).

ssax:read-external-id  port
{sxml.ssax} This procedure parses an ExternalID production.

This procedure is supposed to be called when an ExternalID is expected; that is, the current character must be either #\S or #\P that start correspondingly a SYSTEM or PUBLIC token. This procedure returns the SystemLiteral as a string. A PubidLiteral is disregarded if present.

12.59.3 SSAX higher-level parsers and scanners
They parse productions corresponding to the whole (document) entity or its higher-level pieces (prolog, root element, etc).

ssax:scan-Misc  port
{sxml.ssax} Scan the Misc production in the context:

The following function should be called in the prolog or epilog contexts. In these contexts, whitespaces are completely ignored. The return value from ssax:scan-Misc is either a PI-token, a DECL-token, a START token, or EOF. Comments are ignored and not reported.

ssax:read-char-data  port expect-eof? str-handler seed
{sxml.ssax} This procedure is to read the character content of an XML document or an XML element.

(content ::= (element | CharData | Reference | CDSect | PI | Comment)*}
To be more precise, the procedure reads CharData, expands CDSeect and character entities, and skips comments. The procedure stops at a named reference, EOF, at the beginning of a PI or a start/end tag.

- **port** a port to read
- **expect-eof?** a boolean indicating if EOF is normal, i.e., the character data may be terminated by the EOF. EOF is normal while processing a parsed entity.
- **str-handler** a STR-HANDLER.
- **seed** an argument passed to the first invocation of STR-HANDLER.

The procedure returns two results: seed and token. The seed is the result of the last invocation of str-handler, or the original seed if str-handler was never called.

**Token** can be either an eof-object (this can happen only if expect-eof? was #t), or:
- an xml-token describing a START tag or an END-tag: For a start token, the caller has to finish reading it.
- an xml-token describing the beginning of a PI. It’s up to an application to read or skip through the rest of this PI;
- an xml-token describing a named entity reference.

CDATA sections and character references are expanded inline and never returned. Comments are silently disregarded.

As the XML Recommendation requires, all whitespace in character data must be preserved. However, a CR character (#xD) must be disregarded if it appears before a LF character (#xA), or replaced by a #xA character otherwise. See Secs. 2.10 and 2.11 of the XML Recommendation. See also the canonical XML Recommendation.

### 12.59.4 SSAX Highest-level parsers - XML to SXML

These parsers are a set of syntactic forms to instantiate a SSAX parser. A user can instantiate the parser to do the full validation, or no validation, or any particular validation. The user specifies which PI he wants to be notified about. The user tells what to do with the parsed character and element data. The latter handlers determine if the parsing follows a SAX or a DOM model.

- **ssax:make-pi-parser** my-pi-handlers
  
  My-pi-handlers: An assoc list of pairs (PI-TAG . PI-HANDLER) where PI-TAG is an NCName symbol, the PI target, and PI-HANDLER is a procedure port pi-tag seed where port points to the first symbol after the PI target. The handler should read the rest of the PI up to and including the combination ‘?>’ that terminates the PI. The handler should return a new seed. One of the PI-TAGs may be a symbol *DEFAULT*. The corresponding handler will handle PIs that no other handler will. If the *DEFAULT* PI-TAG is not specified, *ssax:make-pi-parser* will make one, which skips the body of the PI.

  The output of the *ssax:make-pi-parser* is a procedure port pi-tag seed, that will parse the current PI accoding to user-specified handlers.
ssax:make-elem-parser  my-new-level-seed  my-finish-element  
my-char-data-handler  my-pi-handlers

{xml.ssax} Create a parser to parse and process one element, including its character content or child elements. The parser is typically applied to the root element of a document.

my-new-level-seed

procedure  elem-gi  attributes  namespaces  expected-content  seed
where  elem-gi  is  a  RES-NAME  of  the  element  about  to  be  processed. This procedure is to generate the seed to be passed to handlers that process the content of the element.

my-finish-element

procedure  elem-gi  attributes  namespaces  parent-seed  seed
This procedure is called when parsing of  elem-gi  is finished. The  seed  is  the result from the last content parser (or from  my-new-level-seed  if  the  element has the empty content).  Parent-seed  is  the  same  seed  as was  passed  to  my-new-level-seed. The procedure is to generate a seed that will be the result of the element parser.

my-char-data-handler

A  STR-HANDLER.

my-pi-handlers

See  ssax:make-pi-handler  above.

The generated parser is a: procedure  start-tag-head  port  elems  entities  namespaces  preserve-ws?  seed.

The procedure must be called after the start tag token has been read.  Start-tag-head  is  an UNRES-NAME  from  the  start-element  tag.  Elem  is  an  instance  of xml-decl::elems. See ssax:complete-start-tag::preserve-ws?

Faults detected:

VC: XML-Spec.html#elementvalid
WFC: XML-Spec.html#GIMatch

ssax:make-parser  user-handler-tag  user-handler-proc  ...

{xml.ssax} Create an XML parser, an instance of the XML parsing framework. This will be a SAX, a DOM, or a specialized parser depending on the supplied user-handlers.

user-handler-tag is a symbol that identifies a procedural expression that follows the tag. Given below are tags and signatures of the corresponding procedures. Not all tags have to be specified. If some are omitted, reasonable defaults will apply.

tag: DOCTYPE

handler-procedure:  port  docname  systemid  internal-subset?  seed

If  internal-subset?  is  #t,  the  current  position  in  the  port  is  right  after  we  have read  #\[  that  begins  the  internal  DTD  subset.  We  must  finish  reading  of  this subset  before  we  return (or  must  call  skip-internal-subset  if  we  aren’t  interested in reading it). The port  at  exit  must  be  at  the  first symbol  after  the whole DOCTYPE declaration.

The handler-procedure must generate four values:

 elems  entities  namespaces  seed
See  xml-decl::elems  for  elems. It may  be  #f  to  switch  off  the  validation. namespaces  will  typically  contain  USER-PREFIXes  for  selected  URI-SYMBs. The default handler-procedure skips the internal subset, if any, and returns  (values  #f  ()  ()  seed).
tag: **UNDECL-ROOT**
handler-procedure: `elem-gi` `seed`
where `elem-gi` is an UNRES-NAME of the root element. This procedure is called when an XML document under parsing contains **no** DOCTYPE declaration. The handler-procedure, as a DOCTYPE handler procedure above, must generate four values:

```
elems entities namespaces seed
```

The default handler-procedure returns `(values #f () () seed)`.

tag: **DECL-ROOT**
handler-procedure: `elem-gi` `seed`
where `elem-gi` is an UNRES-NAME of the root element. This procedure is called when an XML document under parsing does contain the DOCTYPE declaration. The handler-procedure must generate a new `seed` (and verify that the name of the root element matches the doctype, if the handler so wishes). The default handler-procedure is the identity function.

tag: **NEW-LEVEL-SEED**
handler-procedure: see `ssax:make-elem-parser`, `my-new-level-seed`

tag: **FINISH-ELEMENT**
handler-procedure: see `ssax:make-elem-parser`, `my-finish-element`

tag: **CHAR-DATA-HANDLER**
handler-procedure: see `ssax:make-elem-parser`, `my-char-data-handler`

tag: **PI**
handler-procedure: see `ssax:make-pi-parser`. The default value is `'()`.

The generated parser is a
procedure `PORT SEED`
This procedure parses the document prolog and then exits to an element parser (created by `ssax:make-elem-parser`) to handle the rest.

```
[1] document ::= prolog element Misc*
[22] prolog ::= XMLDecl? Misc* (doctypedecl | Misc*)?
[27] Misc ::= Comment | PI | S

[28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S?
    ('[' (markupdecl | PEReference | S)* ']')? 'S?)? '>'

[29] markupdecl ::= elementdecl | AttlistDecl
    | EntityDecl
    | NotationDecl | PI
    | Comment
```

A few utility procedures that turned out useful.

**ssax:reverse-collect-str**  **fragments**  
{sxml.ssax} given the list of **fragments** (some of which are text strings) reverse the list and concatenate adjacent text strings.

**ssax:reverse-collect-str-drop-ws**  **fragments**  
{sxml.ssax} given the list of **fragments** (some of which are text strings) reverse the list and concatenate adjacent text strings. We also drop "insignificant" whitespace, that is, whitespace in front, behind and between elements. The whitespace that is included in character data is not affected. We use this procedure to "intelligently" drop "insignificant" whitespace in the parsed SXML. If the strict compliance with the XML Recommendation regarding the whitespace is desired, please use the **ssax:reverse-collect-str** procedure instead.
Chapter 12: Library modules - Utilities

12.60 ssax:xml->sxml port namespace-prefix-assig

{ssxml.ssax} This is an instance of a SSAX parser above that returns an SXML representation of the XML document to be read from port. 

Namespace-prefix-assig is a list of (USER-PREFIX . URI-STRING) that assigns USER-PREFIXes to certain namespaces identified by particular URI-STRINGs. It may be an empty list. The procedure returns an SXML tree. The port points out to the first character after the root element.

12.60 sxm1.sxpath - SXML query language

sxm1.sxpath

SXPath is a query language for SXML, an instance of XML Information set (Infoset) in the form of s-expressions.

It is originally written by Oleg Kiselyov, and improved by Dmitry Lizorkin and Kirill Lisovsky. This module also incorporates various procedures written for SXPath by Dmitry Lizorkin and Kirill Lisovsky.

Current version is based on sxpathlib.scm,v 3.915, sxpath.scm,v 1.1, and sxpath-ext.scm,v 1.911.

This manual is mostly derived from the comments in the original source files.

The module consists of three layers.
1. Basic converters and applicators, which provides the means to access and translate SXML tree.
2. High-level query language compiler, which takes abbreviated SXPath and returns a Scheme function that selects a nodeset that satisfies the specified path from the given nodeset.
3. Extension libraries, which implements SXML counterparts to W3C XPath Core Functions Library.

12.60.1 SXPath basic converters and applicators

A converter is a function

\[
\text{type Converter} = \text{Node|Nodeset} \rightarrow \text{Nodeset}
\]

A converter can also play a role of a predicate: in that case, if a converter, applied to a node or a nodeset, yields a non-empty nodeset, the converter-predicate is deemed satisfied. Throughout this file a nil nodeset is equivalent to \#f in denoting a failure.

nodeset? x

{sxm1.sxpath} Returns \#t if given object is a nodeset.

as-nodeset x

{sxm1.sxpath} If x is a nodeset - returns it as is, otherwise wrap it in a list.

sxm1:element? obj

{sxm1.sxpath} Predicate which returns \#t if obj is SXML element, otherwise returns \#f.

ntype-names?? crit

{sxm1.sxpath} The function ntype-names?? takes a list of acceptable node names as a criterion and returns a function, which, when applied to a node, will return \#t if the node name is present in criterion list and \#f otherwise.

ntype-names?? :: ListOfNames -> Node -> Boolean
**ntype?? ** [Function]  
\{sxml.xpath\} The function ntype?? takes a type criterion and returns a function, which, when applied to a node, will tell if the node satisfies the test.

\[\text{ntype??} :: \text{Crit} \rightarrow \text{Node} \rightarrow \text{Boolean}\]

The criterion crit is one of the following symbols:

- **id**  tests if the Node has the right name (id)
- **@**  tests if the Node is an attributes-list.
- ***text***  tests if the Node is a text node.
- ***data***  tests if the Node is a data node (text, number, boolean, etc., but not pair).
- ***PI***  tests if the Node is a PI node.
- ***COMMENT***  tests if the Node is a COMMENT node.
- ***ENTITY***  tests if the Node is an ENTITY node.
- ***any***  #t for any type of Node.

**ntype-namespace-id?? ns-id** [Function]  
\{sxml.xpath\} This function takes a namespace-id, and returns a predicate Node \(\rightarrow\) Boolean, which is #t for nodes with this very namespace-id. ns-id is a string.

\[(\text{ntype-namespace-id??} \#f)\] will be #t for nodes with non-qualified names.

**sxml:invert pred** [Function]  
\{sxml.xpath\} This function takes a predicate and returns it inverted. That is if the given predicate yields #f or '() the inverted one yields the given node (#t) and vice versa.

**node-eq? other** [Function]  
\{sxml.xpath\} Curried equivalence converter-predicates, i.e.

\[(\text{node-eq? a}) b \equiv (\text{eq? a} b)\]

\[(\text{node-equal? a}) b \equiv (\text{equal? a} b)\]

**node-pos n** [Function]  
\{sxml.xpath\}

\[\text{node-pos\:} N \rightarrow \text{Nodeset} \rightarrow \text{Nodeset}, \text{or} \]
\[\text{node-pos\:} N \rightarrow \text{Converter}\]

Select the N’th element of a Nodeset and return as a singular Nodeset; Return an empty nodeset if the Nth element does not exist. ((node-pos 1) Nodeset) selects the node at the head of the Nodeset, if exists; ((node-pos 2) Nodeset) selects the Node after that, if exists. N can also be a negative number: in that case the node is picked from the tail of the list. ((node-pos -1) Nodeset) selects the last node of a non-empty nodeset; ((node-pos -2) Nodeset) selects the last but one node, if exists.

**sxml:filter pred?** [Function]  
\{sxml.xpath\}

\[\text{filter\:} \rightarrow \text{Converter} \rightarrow \text{Converter}\]

A filter applicator, which introduces a filtering context. The argument converter is considered a predicate, with either #f or nil result meaning failure.
Chapter 12: Library modules - Utilities

**take-until** *pred?*  
{sxml.xpath}
  
  take-until:: Converter -> Converter, or  
  take-until:: Pred -> Node|Nodeset -> Nodeset

Given a converter-predicate and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but #f or nil. Return the elements of the input nodeset that have been processed till that moment (that is, which fail the predicate). **take-until** is a variation of the filter above: **take-until** passes elements of an ordered input set till (but not including) the first element that satisfies the predicate. The nodeset returned by ((take-until (not pred)) nset) is a subset – to be more precise, a prefix – of the nodeset returned by ((filter pred) nset).

**take-after** *pred?*  
{sxml.xpath}
  
  take-after:: Converter -> Converter, or  
  take-after:: Pred -> Node|Nodeset -> Nodeset

Given a converter-predicate and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but #f or nil. Return the elements of the input nodeset that have not been processed: that is, return the elements of the input nodeset that follow the first element that satisfied the predicate. **take-after** along with **take-until** partition an input nodeset into three parts: the first element that satisfies a predicate, all preceding elements and all following elements.

**map-union** *proc lst*  
{sxml.xpath} Apply proc to each element of lst and return the list of results. If proc returns a nodeset, splice it into the result.

From another point of view, **map-union** is a function Converter->Converter, which places an argument-converter in a joining context.

**node-reverse** *node-or-nodeset*  
{sxml.xpath}
  
  node-reverse :: Converter, or  
  node-reverse:: Node|Nodeset -> Nodeset

Reverses the order of nodes in the nodeset. This basic converter is needed to implement a reverse document order (see the XPath Recommendation).

**node-trace** *title*  
{sxml.xpath}
  
  node-trace:: String -> Converter

(node-trace title) is an identity converter. In addition it prints out a node or nodeset it is applied to, prefixed with the 'title'. This converter is very useful for debugging.

What follow are Converter combinators, higher-order functions that transmogrify a converter or glue a sequence of converters into a single, non-trivial converter. The goal is to arrive at converters that correspond to XPath location paths.

From a different point of view, a combinator is a fixed, named pattern of applying converters. Given below is a complete set of such patterns that together implement XPath location path specification. As it turns out, all these combinators can be built from a small number of basic blocks: regular functional composition, map-union and filter applicators, and the nodeset union.
select-kids test-pred?  {sxml.sxpath}

select-kids:: Pred -> Node -> Nodeset

Given a Node, return an (ordered) subset its children that satisfy the Pred (a converter, actually).

select-kids:: Pred -> Nodeset -> Nodeset

The same as above, but select among children of all the nodes in the Nodeset.

node-self pred  {sxml.sxpath}

node-self:: Pred -> Node -> Nodeset, or
node-self:: Converter -> Converter

Similar to select-kids but apply to the Node itself rather than to its children. The resulting Nodeset will contain either one component, or will be empty (if the Node failed the Pred).

node-join . selectors  {sxml.sxpath}

node-join:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-join:: [Converter] -> Converter

join the sequence of location steps or paths as described in the title comments above.

node-reduce . converters  {sxml.sxpath}

node-reduce:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-reduce:: [Converter] -> Converter

A regular functional composition of converters. From a different point of view, ((apply node-reduce converters) nodeset) is equivalent to (foldl apply nodeset converters) i.e., folding, or reducing, a list of converters with the nodeset as a seed.

node-or . converters  {sxml.sxpath}

node-or:: [Converter] -> Converter

This combinator applies all converters to a given node and produces the union of their results. This combinator corresponds to a union, '|' operation for XPath location paths.

node-closure test-pred?  {sxml.sxpath}

node-closure:: Converter -> Converter

Select all descendants of a node that satisfy a converter-predicate. This combinator is similar to select-kids but applies to grand... children as well. This combinator implements the "descendant::" XPath axis. Conceptually, this combinator can be expressed as

(define (node-closure f)
  (node-or
   (select-kids f)
   (node-reduce (select-kids (ntype?? '*)) (node-closure f)))))

This definition, as written, looks somewhat like a fixpoint, and it will run forever. It is obvious however that sooner or later (select-kids (ntype?? '*)) will return an empty nodeset. At this point further iterations will no longer affect the result and can be stopped.
12.60.2 SXPath query language

**sxpath abbrpath . ns-binding**

(Evaluates an abbreviated SXPath. Returns a procedure that when applied on a node or nodeset will return a nodeset matching the given path.

\[sxpath:: AbbrPath -> Converter, or\]
\[sxpath:: AbbrPath -> Node|Nodeset -> Nodeset\]

**AbbrPath** is a list or a string. If it is a list, it is translated to the full SXPath according to the following rewriting rules. More informal explanation follows shortly. If it is a string, it is an XPath query.

Note that these are abstract rules to show how it works, and not the running code examples. The nonterminals sxpath1 and sxpathr don’t exist as APIs. The term txpath is an internal function that interprets XPath query given as a string.

- \((sxpath () -> (node-join))\)
- \((sxpath (path-component ...)) -> (node-join (sxpath1 path-component) (sxpath (....)))\)
- \((sxpath1 //) -> (node-or (node-self (ntype?? '*any*)) (node-closure (ntype?? '*any*))))\)
- \((sxpath1 '(equal? x)) -> (select-kids (node-equal? x))\)
- \((sxpath1 '(eq? x)) -> (select-kids (node-eq? x))\)
- \((sxpath1 '(or@ ...)) -> (select-kids (ntype-names?? (cdr '(or@ ...))))\)
- \((sxpath1 '(not@ ...)) -> (select-kids (sxxml:invert (ntype-names?? (cdr '(not@ ...))))))\)
- \((sxpath1 '(ns-id:* x)) -> (select-kids (ntype-namespace-id?? x))\)
- \((sxpath1 ?symbol) -> (select-kids (ntype?? ?symbol))\)
- \((sxpath1 ?string) -> (txpath ?string))\)
- \((sxpath1 procedure) -> procedure\)
- \((sxpath1 '(@(symbol ...)) -> (sxpath1 '(@(?symbol ...)))\)

SXPath in its simplest form is a list of path components. The result procedure will follow the same path and return the matching node list. For example \((one two three)\) will find element one then two inside it and three inside element two. The equivalent XPath would be one/two/three.

There are a few special path components (see ntype?? for the complete list):

- * matches an element node.
- // matches any one or many consecutive path components.
- *text* matches a text node (text() in XPath).
- *data* matches any data node (e.g. text, number, boolean, etc., but not pair).
- @ selects the attribute list node.

A path component could be a list in one of these forms:

- \((equal? x)\) matches if the node under examination matches x using node-equal?
(eq? x) matches if the node under examination matches x using node-eq?
(or@ ...) matches if the element name is one of the specified symbols.
(not@ ...) matches if the element name is not one of the specified symbols.
(ns-id:* x) matches the node if it’s with namespace x
(<path> n) matches the n-th node matching same path component. n starts from 1. Negative numbers start from the end of the node list backward. This is path[n] syntax in XPath.
((<path> (predicate)...) matches a path component path and (xpath (predicate)...) on those nodes are not empty. This is path[predicate...] syntax in XPath.

If the path component is a string, it is interpreted as an XPath query string.
If the path component is a procedure, the procedure takes three arguments: the nodeset being examined, the root node and the variable bindings.
The root node is usually the entire sxml being applied. However if you apply the result xpath procedure with two arguments, root-node will be the second argument.

When applied with three arguments, the variable bindings are the third one. This lets you pass arguments to the procedure.

;; select all <book> elements whose style attribute value is equal to
;; the <bookstore> element’s specialty attribute value.
(xpath "/book[@specialty= @style]") ;; a similar query but this time make sure specialty of _all_
;; bookstores is matched
(let ([match-specialty
   (lambda (node root var-binding)
        (let ([style (car ((xpath '(@ style *text*)) node))]
               [all-specialty ((xpath '(bookstore @ specialty *text*)) node)])
            (fold (lambda (specialty last-result)
                   (and last-result (string=? style specialty])))
               #t
               all-specialty)))))]
   (xpath ‘(/// (book ,match-specialty))))

;; select all <bookstore> elements that are inside top-level <book>
;; element
(xpath 'book bookstore)) ;; select all <bookstore> elements from anywhere
(xpath ‘(/// bookstore)) ;; select attribute "name" in the top-level <book> element
(xpath 'book @ name)) ;; select all <bookstore> and <bookshop> elements that are inside
;; top-level <book> element
(xpath 'book (or@ bookstore bookshop))) ;; select all elements except <movie> that are inside top-level <book>
;; element
(xpath 'book (not@ movie @)) ;; select the attribute "name" of the second <bookstore> element
(xpath 'book (bookstore 2) @ name)) ;; select the attribute "name" of all <bookstore> elements that has
;; attribute "recommended"
(xpath 'book (bookstore (0 recommended)) @ name)) ;; select the attribute "name" of all <bookstore> elements whose
;; "rating" attribute is 3
(xpath 'book (bookstore (0 rating (eq? 3)) @ name))
12.60.3 SXPath extension

SXML counterparts to W3C XPath Core Functions Library.

sxml:string object
{sxpath} The counterpart to XPath string function (section 4.2 XPath Rec.) Converts a given object to a string. NOTE:
1. When converting a nodeset - a document order is not preserved
2. number->string function returns the result in a form which is slightly different from XPath Rec. specification

sxml:boolean object
{sxpath} The counterpart to XPath boolean function (section 4.3 XPath Rec.) Converts its argument to a boolean.

sxml:number obj
{sxpath} The counterpart to XPath number function (section 4.4 XPath Rec.) Converts its argument to a number NOTE:
1. The argument is not optional (yet?).
2. string->number conversion is not IEEE 754 round-to-nearest.
3. NaN is represented as 0.

sxml:string-value node
{sxpath} Returns a string value for a given node in accordance to XPath Rec. 5.1 - 5.7

sxml:node? node
{sxpath} According to XPath specification 2.3, this test is true for any XPath node. For SXML auxiliary lists and lists of attributes has to be excluded.

sxml:attr-list obj
{sxpath} Returns the list of attributes for a given SXML node. Empty list is returned if the given node is not an element, or if it has no list of attributes
**sxml:id** *id-index*  
{sxml.sxpath} Select SXML element by its unique IDs. (XPath Rec. 4.1) Returns a converter that takes *object*, which is a nodeset or a datatype which can be converted to a string by means of a 'string' function.

*id-index* is \((\text{id-value . element} \text{id-value . element})\ldots\).  
This index is used for selection of an element by its unique ID.

Comparators for XPath objects:

**sxml:equal?** *a b*  
**sxml:not-equal?** *a b*  
{sxml.sxpath} Counterparts of XPath equality operations: =, !=, using default equality tests.

**sxml:relational-cmp** *op*  
{sxml.sxpath} Creates a relational operation ( <, >, <=, >= ) for two XPath objects. *op* is comparison procedure: <, >, <= or >=.

XPath axises. An order in resulting nodeset is preserved.

**sxml:attribute** *test-pred?*  
{sxml.sxpath} Attribute axis.

**sxml:child** *test-pred?*  
{sxml.sxpath} Child axis. This function is similar to 'select-kids', but it returns an empty child-list for PI, Comment and Entity nodes.

**sxml:parent** *test-pred?*  
{sxml.sxpath} Parent axis.

Given a predicate, it returns a function RootNode -> Converter which yields a node -> parent converter then applied to a rootnode.  
Thus, such a converter may be constructed using \((\text{sxml:parent test-pred})\text{ rootnode}) and returns a parent of a node it is applied to. If applied to a nodeset, it returns the list of parents of nodes in the nodeset. The rootnode does not have to be the root node of the whole SXML tree – it may be a root node of a branch of interest. The parent:: axis can be used with any SXML node.

**sxml:ancestor** *test-pred?*  
{sxml.sxpath} Ancestor axis

**sxml:ancestor-or-self** *test-pred?*  
{sxml.sxpath} Ancestor-or-self axis

**sxml:descendant** *test-pred?*  
{sxml.sxpath} Descendant axis

**sxml:descendant-or-self** *test-pred?*  
{sxml.sxpath} Descendant-or-self axis

**sxml:following** *test-pred?*  
{sxml.sxpath} Following axis
sxml:following-sibling test-pred? [Function]
    {sxml.sxpath} Following-sibling axis

sxml:namespace test-pred? [Function]
    {sxml.sxpath} Namespace axis

sxml:preceding test-pred? [Function]
    {sxml.sxpath} Preceding axis

sxml:preceding-sibling test-pred? [Function]
    {sxml.sxpath} Preceding-sibling axis

Popular shortcuts:

sxml:child-nodes nodeset [Function]
    {sxml.sxpath}
    ((sxml:child sxml:node?) nodeset)

sxml:child-elements nodeset [Function]
    {sxml.sxpath}
    ((select-kids sxml:element?) nodeset)

12.61 sxml.tools - Manipulating SXML structure

sxml.tools [Module]
    This module is a port of Kirill Lisofsky’s sxml-tools, a collection of convenient procedures
    that work on SXML structure. The current version is derived from sxml-tools CVS revision
    3.13.

    The manual entry is mainly derived from the comments in the original source code.

12.61.1 SXML predicates

sxml:empty-element? obj [Function]
    {sxml.tools} A predicate which returns #t if given element obj is empty. Empty element
    has no nested elements, text nodes, PIs, Comments or entities but it may contain attributes
    or namespace-id. It is a SXML counterpart of XML empty-element.

sxml:shallow-normalized? obj [Function]
    {sxml.tools} Returns #t if the given obj is shallow-normalized SXML element. The element
    itself has to be normalized but its nested elements are not tested.

sxml:normalized? obj [Function]
    {sxml.tools} Returns #t if the given obj is normalized SXML element. The element itself
    and all its nested elements have to be normalised.

sxml:shallow-minimized? obj [Function]
    {sxml.tools} Returns #t if the given obj is shallow-minimized SXML element. The element
    itself has to be minimised but its nested elements are not tested.

sxml:minimized? obj [Function]
    {sxml.tools} Returns #t if the given obj is minimized SXML element. The element itself
    and all its nested elements have to be minimised.
### 12.61.2 SXML accessors

**sxml:name** *obj*  
{**sxml.tools**} Returns a name of a given SXML node. It’s just an alias of **car**, but introduced for the sake of encapsulation.

**sxml:element-name** *obj*  
{**sxml.tools**} A version of **sxml:name**, which returns **#f** if the given *obj* is not a SXML element. Otherwise returns its name.

**sxml:node-name** *obj*  
{**sxml.tools**} Safe version of **sxml:name**, which returns **#f** if the given *obj* is not a SXML node. Otherwise returns its name.

**sxml:ncname** *obj*  
{**sxml.tools**} Returns Local Part of Qualified Name (Namespaces in XML production [6]) for given *obj*, which is ":"-separated suffix of its Qualified Name. If a name of a node given is NCName (Namespaces in XML production [4]), then it is returned as is. Please note that while SXML name is a symbol this function returns a string.

**sxml:name->ns-id** *sxml-name*  
{**sxml.tools**} Returns namespace-id part of given name, or **#f** if it’s LocalName

**sxml:content** *obj*  
{**sxml.tools**} Returns the content of given SXML element or nodeset (just text and element nodes) representing it as a list of strings and nested elements in document order. This list is empty if *obj* is empty element or empty list.

**sxml:content-raw** *obj*  
{**sxml.tools**} Returns all the content of normalized SXML element except attr-list and aux-list. Thus it includes PI, COMMENT and ENTITY nodes as well as TEXT and ELEMENT nodes returned by **sxml:content**. Returns a list of nodes in document order or empty list if *obj* is empty element or empty list. This function is faster than **sxml:content**.

In SXML normal form, an element is represented by a list as this:

```
(name attr-list aux-list content ...)
```

where attr-list is a list beginning with @, and aux-list is a list beginning with @@.

In the minimized form, Aux-list can be omitted when it is empty. Attr-list can be omitted when it is empty and aux-list is absent.

The following procedures extract attr-list and aux-list.

**sxml:attr-list-node** *obj*  
{**sxml.tools**} Returns attr-list for a given *obj*, or **#f** if it is absent

**sxml:attr-as-list** *obj*  
{**sxml.tools**} Returns attr-list wrapped in list, or "((@))" if it is absent and aux-list is present, or "()" if both lists are absent.

**sxml:aux-list-node** *obj*  
{**sxml.tools**} Returns aux-list for a given *obj*, or **#f** if it is absent.

**sxml:aux-as-list** *obj*  
{**sxml.tools**} Returns aux-list wrapped in list, or "()" if it is absent.
**Function**

`sxml:attr-list-u obj`

{`sxml.tools`} Returns the list of attributes for given element or nodeset. Analog of `((sxpath '(@ *)) obj)`. Empty list is returned if there is no list of attributes.

The `-u` suffix indicates it can be used for non-normalized SXML node. ('u' stands for 'universal').

**Function**

`sxml:aux-list obj`

{`sxml.tools`} Returns the list of auxiliary nodes for given element or nodeset. Analog of `((sxpath '(@@ *)) obj)`. Empty list is returned if a list of auxiliary nodes is absent.

**Function**

`sxml:aux-list-u obj`

{`sxml.tools`} Returns the list of auxiliary nodes for given element or nodeset. Analog of `((sxpath '(@@ *)) obj)`. Empty list is returned if a list of auxiliary nodes is absent.

The `-u` suffix indicates it can be used for non-normalized SXML node. ('u' stands for 'universal').

**Function**

`sxml:aux-node obj aux-name`

{`sxml.tools`} Return the first aux-node with `aux-name` given in SXML element `obj` or `#f` is such a node is absent. Note: it returns just the first node found even if multiple nodes are present, so it’s mostly intended for nodes with unique names.

**Function**

`sxml:aux-nodes obj aux-name`

{`sxml.tools`} Return a list of aux-node with `aux-name` given in SXML element `obj` or `()` if such a node is absent.

**Function**

`sxml:attr obj attr-name`

{`sxml.tools`} Accessor for an attribute `attr-name` of given SXML element `obj`. It returns: the value of the attribute if the attribute is present, or `#f` if there is no such an attribute in the given element.

**Function**

`sxml:num-attr obj attr-name`

{`sxml.tools`} Accessor for a numerical attribute `attr-name` of given SXML element `obj`. It returns: a value of the attribute as the attribute as a number if the attribute is present and its value may be converted to number using `string->number`, or `#f` if there is no such an attribute in the given element or its value can’t be converted to a number.

**Function**

`sxml:attr-u obj attr-name`

{`sxml.tools`} Accessor for an attribute `attr-name` of given SXML element `obj` which may also be an attributes-list or nodeset (usually content of SXML element).

It returns: the value of the attribute if the attribute is present, or `#f` if there is no such an attribute in the given element.

The `-u` suffix indicates it can be used for non-normalized SXML node. ('u' stands for 'universal').

**Function**

`sxml:ns-list obj`

{`sxml.tools`} Returns the list of namespaces for given element. Analog of `((sxpath '(@@ *NAMESPACES* *)) obj)` Empty list is returned if there is no list of namespaces.

**Function**

`sxml:ns-id->nodes obj namespace-id`

{`sxml.tools`} Returns the list of namespace-assoc’s for given `namespace-id` in SXML element `obj`. Analog of `((sxpath '(@@ *NAMESPACES* namespace-id) obj)` Empty list is returned if there is no namespace-assoc with `namespace-id` given.
Chapter 12: Library modules - Utilities

**Function**

`sxml:ns-id->uri` obj namespace-id

{`sxml.tools`} Returns a URI for namespace-id given, or `#f` if there is no namespace-assoc with namespace-id given.

**Function**

`sxml:ns-uri->id` obj uri

{`sxml.tools`} Returns a namespace-id for namespace URI given.

**Function**

`sxml:ns-id` ns-assoc

{`sxml.tools`} Returns namespace-id for given namespace-assoc list.

**Function**

`sxml:ns-uri` ns-assoc

{`sxml.tools`} Returns URI for given namespace-assoc list.

**Function**

`sxml:ns-prefix` ns-assoc

{`sxml.tools`} It returns namespace prefix for given namespace-assoc list. Original (as in XML document) prefix for namespace-id given has to be stored as the third element in namespace-assoc list if it is different from namespace-id. If original prefix is omitted in namespace-assoc then namespace-id is used instead.

### 12.61.3 SXML modifiers

Constructors and mutators for normalized SXML data. These functions are optimized for normalized SXML data. They are not applicable to arbitrary non-normalized SXML data.

Most of the functions are provided in two variants:

1. side-effect intended functions for linear update of given elements. Their names are ended with exclamation mark. Note that the returned value of this variant is unspecified, unless explicitly noted. An example: `sxml:change-content!`.

2. pure functions without side-effects which return modified elements. An example: `sxml:change-content`.

**Function**

`sxml:change-content` obj new-content

**Function**

`sxml:change-content!` obj new-content

{`sxml.tools`} Change the content of given SXML element to `new-content`. If `new-content` is an empty list then the `obj` is transformed to an empty element. The resulting SXML element is normalized.

**Function**

`sxml:change-attrlist` obj new-attrlist

**Function**

`sxml:change-attrlist!` obj new-attrlist

{`sxml.tools`} The resulting SXML element is normalized. If `new-attrlist` is empty, the `cadr` of `obj` is `@`.

**Function**

`sxml:change-name` obj new-name

**Function**

`sxml:change-name!` obj new-name

{`sxml.tools`} Change a name of SXML element destructively.

**Function**

`sxml:add-attr` obj attr

{`sxml.tools`} Returns SXML element `obj` with attribute `attr` added, or `#f` if the attribute with given name already exists. `attr` is `(attr-name attr-value)`. Pure functional counterpart to `sxml:add-attr!`.

**Function**

`sxml:add-attr!` obj attr

{`sxml.tools`} Add an attribute `attr` for an element `obj`. Returns `#f` if the attribute with given name already exists. The resulting SXML node is normalized. Linear update counterpart to `sxml:add-attr`. 


Chapter 12: Library modules - Utilities

12.61.4 SXPath auxiliary utilities

These are convenience utilities to extend SXPath functionalities.

12.61.5 SXML to markup conversion

Procedures to generate XML or HTML marked up text from SXML. For more advanced conversion, see the SXML serializer (Section 12.62 [Serializing XML and HTML from SXML], page 983).
Chapter 12: Library modules - Utilities

**sxml:attr->xml** attr
{sxm.tools} Creates the XML markup for attributes.

**sxml:string->xml** string
{sxm.tools} Return a string or a list of strings where all the occurrences of characters <, >, &, " or ' in a given string are replaced by corresponding character entity references. See also sxml:string->html.

**sxml:sxml->xml** tree
{sxm.tools} A version of dispatch-node specialized and optimized for SXML->XML transformation.

**sxml:attr->html** attr
{sxm.tools} Creates the HTML markup for attributes.

**sxml:string->html** string
{sxm.tools} Given a string, check to make sure it does not contain characters <, >, &, " that require encoding. See also html-escape-string in Section 12.70 [Simple HTML document construction], page 1002.

**sxml:non-terminated-html-tag?** tag
{sxm.tools} This predicate yields #t for "non-terminated" HTML 4.0 tags.

**sxml:sxml->html** tree
{sxm.tools} A version of dispatch-node specialized and optimized for SXML->HTML transformation.

### 12.62 sxml.serializer - Serializing XML and HTML from SXML

**sxml.serializer**
This module contains a full-featured serializer from SXML into XML and HTML, partially conforming to XSLT 2.0 and XQuery 1.0 Serialization (http://www.w3.org/TR/2005/CR-xslt-xquery-serialization-20051103/). It’s more powerful than sxml:sxml->xml and sxml:sxml->html from sxm.tools.

The manual entry is mainly derived from the comments in the original source code.

### 12.62.1 Simple SXML serializing

The SXML serializer provides some convenient high-level converters which should be enough for most tasks.

**srl:sxml->xml** sxml-obj :optional port-or-filename
{sxm.serializer} Serializes the sxml-obj into XML, with indentation to facilitate readability by a human.

If port-or-filename is not supplied, the functions return a string that contains the serialized representation of the sxml-obj.

If port-or-filename is supplied and is a port, the functions write the serialized representation of sxml-obj to this port and return an unspecified result.

If port-or-filename is supplied and is a string, this string is treated as an output filename, the serialized representation of sxml-obj is written to that filename and an unspecified result is returned. If a file with the given name already exists, the effect is unspecified.

**srl:sxml->xml-noindent** sxml-obj :optional port-or-filename
{sxm.serializer} Serializes the sxml-obj into XML, without indentation.

Argument port-or-filename works like described in srl:sxml->xml.
Chapter 12: Library modules - Utilities

**srl:sxml->html**

```
{sxml.serializer} Serializes the sxml-obj into HTML, with indentation to facilitate readability by a human.
```

Argument *port-or-filename* works like described in *srl:sxml->xml*.

**srl:sxml->html-noindent**

```
{sxml.serializer} Serializes the sxml-obj into HTML, without indentation.
```

Argument *port-or-filename* works like described in *srl:sxml->xml*.

### 12.62.2 Custom SXML serializing

These functions provide full access to all configuration parameters of the XML serializer.

**srl:parameterizable**

```
{sxml.serializer} Generalized serialization procedure, parameterizable with all the serialization parameters supported by this implementation.
```

*sxml-obj* - an SXML object to serialize

*port-or-filename* - either #f, a port or a string; works like in *srl:sxml->xml* (Section 12.62.1 [Simple SXML serializing], page 983).

*params* - each parameter is a cons of param-name (a symbol) and param-value. The available parameter names and their values are described below:

- **method** - Either the symbol *xml* or *html*. For a detailed explanation of the difference between XML and HTML methods, see XSLT 2.0 and XQuery 1.0 Serialization (http://www.w3.org/TR/2005/CR-xslt-xquery-serialization-20051103/).
- **indent** - Whether the output XML should include whitespace for human readability (#t or #f). You can also supply a string, which will be used as the indentation unit.
- **omit-xml-declaration** - Whether the XML declaration should be omitted. Default: #t.
- **standalone** - Whether to define the XML document as standalone in the XML declaration. Should be one of the symbols *yes*, *no* or *omit*, the later causing standalone declaration to be suppressed. Default: *omit*.
- **version** - The XML version used in the declaration. A string or a number. Default: "1.0".
- **cdata-section-elements** - A list of SXML element names (as symbols). The contents of those elements will be escaped as CDATA sections.
- **ns-prefix-assig** - A list of (cons prefix namespace-uri), where each prefix is a symbol and each namespace-uri a string. Will serialize the given namespaces with the corresponding prefixes.

ATTENTION: If a parameter name is unexpected or a parameter value is ill-formed, the parameter is silently ignored!

Example usage:

```scheme
(srl:parameterizable
 '((tag (@ (attr "value")) (nested "text node") (empty))
 (current-output-port)
 'method . xml) ; XML output method is used by default
 'indent . "\t") ; use a single tabulation to indent
 'omit-xml-declaration . #f) ; add XML declaration
 'standalone . yes) ; denote a standalone XML document
 'version . "1.0") ; XML version
param ::= (cons param-name param-value)
param-name ::= symbol
```
Chapter 12: Library modules - Utilities

```
cdata-section-elements
value ::= (listof sxml-elem-name)
sxml-elem-name ::= symbol

indent
value ::= 'yes | #t | 'no | #f | whitespace-string

method
value ::= 'xml | 'html

ns-prefix-assig
value ::= (listof (cons prefix namespace-uri))
prefix ::= symbol
namespace-uri ::= string

omit-xml-declaration
value ::= 'yes | #t | 'no | #f

standalone
value ::= 'yes | #t | 'no | #f | 'omit

version
value ::= string | number
```

**srl:sxml->string**  
`sxml-obj cdata-section-elements indent method`  
*ns-prefix-assig omit-xml-declaration? standalone version*  
`{sxml.serializer}` Same as **srl:parameterizable** returning a string and without the overhead of parsing parameters. This function interface may change in future versions of the library.

**srl:display-sxml**  
`sxml-obj port-or-filename cdata-section-elements`  
*indent method ns-prefix-assig omit-xml-declaration? standalone version*  
`{sxml.serializer}` Same as **srl:parameterizable** writing output to `port-or-filename` and without the overhead of parsing parameters. This function interface may change in future versions of the library.

### 12.63 text.console - Text terminal control

**text.console**  
This module provides a simple interface for character terminal control. Currently we support vt100 compatible terminals and Windows console.

This module doesn’t depend on external library such as `curses` and works with Gauche alone, but what it can do is limited; for example, you can’t get an event when shift key alone is pressed. For finer controls, you need some extension libraries.

For an example of the features in this module, see `snake.scm` in the examples directory of Gauche source distribution.

**Console objects**

**<vt100>**  
`{text.console}` Represents a vt100-compatible terminal. An instance of this class can be passed to the “console” argument of the following generic functions.
iport  [Instance Variable of <vt100>]
Input port connected to the terminal. The default value is the standard input port.

oport  [Instance Variable of <vt100>]
Output port connected to the terminal. The default value is the standard output port.

input-delay  [Instance Variable of <vt100>]
The terminal sends back special keys encoded in an input escape sequence. In order to
distinguish such keys from the actual ESC key, we time the input—if the subsequent
input doesn’t come within input-delay microseconds, we interpret the input as individual
keystroke, rather than a part of an escape sequence. The default value is 1000 (1ms).

<windows-console>  [Class]
Represents Windows console. This class is defined on all platforms, but its useful methods
are only available on Windows-native runtime.
It doesn’t have public slots.

The application has to check the runtime to see what kind of console is available. A suggested
flow is as follows.

• If has-windows-console? returns true, create <windows-console> instance. You don’t
need cond-expand; has-windows-console? returns #f on non-Windows platforms.
• Check the environment variable TERM. If it is set and satisfies vt100-compatible?, you can
create <vt100> instance. (Note: It is possible that you end up using <vt100> console on
Windows; e.g. gosh running on MSYS shell.)
• Otherwise, console isn’t available.

The following procedure packages this flow.

make-default-console :key if-not-available
{text.console} Determines a suitable console class of the running process and returns its
instance.
If no suitable console is available, the behavior depends on the if-not-available keyword argu-
ment. If it is :error, which is default, an error is signalled. If it is #f, the procedure returns
#f.

vt100-compatible? string
{text.console} Given the string value of the environment variable TERM, returns #t if the
terminal can be handled by <vt100> console, #f otherwise.

Console control

call-with-console console proc :key mode
{text.console} Takes over the control of the console, and calls proc with console as the only
argument. The console is set to the mode, which must be a symbol with-terminal-mode
accepts: raw, rare or cooked. By default the console is set to rare mode, which turn off the
echoing and passes most of keystrokes to the program, but it intercepts terminal controls (like
Ctrl-C for interrupt and Ctrl-Z for suspend; the actual key depends on terminal settings,
though. See Section 9.33 [Terminal control], page 518, for the details.)
If proc raises an unhandled error, this generic function resets the terminal mode before
returning. It does not clear the screen.

putch console char
{text.console} Display a character at the current cursor position, and move the current
cursor position.
Chapter 12: Library modules - Utilities

`putstr` `console string`  
{text.console} Display a string from the current cursor position, and move the current cursor position.

`beep` `console`  
{text.console} Ring the beep, or flash the screen (visible bell) if possible.

`getch` `console`  
{text.console} Fetch a keypress from the console. This blocks until any key is pressed.

The return value may be one of the following values:

A character
A key for the character is pressed. It may be a control code if the control key is pressed with the key; that is, if the user presses Ctrl-A, 
\textbackslash x01
will be returned.

A symbol
Indicates a special key; the following keys are supported: KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT, KEY_HOME, KEY_END, KEY_INS, KEY_DEL, KEY_PGDN, KEY_PGUP, KEY_F1, KEY_F2, KEY_F3, KEY_F4, KEY_F5, KEY_F6, KEY_F7, KEY_F8, KEY_F9, KEY_F10, KEY_F11, KEY_F12. (Note: DELETE key is usually mapped to 
\textbackslash x7f,
but it depends on the terminal).

A list of symbol ALT and a character.
Indicates the character key is pressed with Alt key. For example, if the user presses Alt-a, (ALT #\a) is returned.

EOF
Indicates the input is closed somehow.

Modifier keys except ALT are not treated separately but included in the returned keycode. Assuming CAPSLOCK is off, if the user presses a, Shift+a, and Ctrl+a, the returned value is 
\textbackslash a, \textbackslash A and \textbackslash x01, respectively. Ctrl+Shift+a can’t be distinguished from Ctrl+a. ALT+a, ALT+Shift+a, and ALT+Ctrl+a will be (ALT #\a), (ALT #\A) and (ALT #\x01), respectively.

`chready?` `console`  
{text.console} Returns true if there’s a key sequence to be read in the console’s input.

`query-cursor-position` `console`  
{text.console} Returns two values, the current cursor’s x and y position. The top-left corner is (0,0).

`move-cursor-to` `console row column`  
{text.console} Move cursor to the specified position. The top-left corner is (0,0).

`reset-terminal` `console`  
{text.console} Reset terminal. Usually this sets the character attributes to the default, clears the screen, and moves the cursor to (0,0).

`clear-screen` `console`  
{text.console} Clear entire screen.

`clear-to-eol` `console`  
{text.console} Clear characters from the current cursor position to the end of the line.

`clear-to-eos` `console`  
{text.console} Clear characters from the current cursor position to the end of the screen.

`hide-cursor` `console`  
{text.console} Hide/show the cursor.

`show-cursor` `console`  
{text.console} Hide/show the cursor.
cursor-down/scroll-up console
{text.console} If the cursor is at the bottom line of the screen, scroll up the contents and clear the bottom line; the cursor stays the same position. If the cursor is not at the bottom line of the screen, move the cursor down.

cursor-up/scroll-down console
{text.console} If the cursor is at the top line of the screen, scroll down the contents and clear the top line; the cursor stays the same position. If the cursor is not at the top line of the screen, move the cursor up.

query-screen-size console
{text.console} Returns two values, the width and height of the screen.
Note: This may affect what’s shown in the console. It is recommended that you only call this before redrawing the entire screen and save the result.

set-character-attribute console spec
{text.console} Set the console so that the subsequent characters will be written with attributes specified by spec.

The character attributes spec is a list in the following format:

\( (<\text{fgcolor}> \ [<\text{bgcolor}> . \ <\text{option}> . . .]) \)

where:

- \(<\text{fgcolor}>\) : \(<\text{color}>\) | #f ; #f means default
- \(<\text{bgcolor}>\) : \(<\text{color}>\) | #f
- \(<\text{color}>\) : black | red | green | yellow | blue | magenta | cyan | white
- \(<\text{option}>\) : bright | reverse | underscore

For example, you can set characters to be written in red with black background and underscore, you can call:

\( (\text{set-character-attribute con '}(\text{red black underscore})) \)

That the options may seem rather limited in the age of full-color bitmap displays. That’s what it used to be, young lads.

reset-character-attribute console
{text.console} Reset character attributes to the default.

with-character-attribute console attrs thunk
{text.console} Sets the console’s attributes to attrs and calls thunk, then restores the attributes. Even if thunk throws an error, attributes are restored.

Note: You should be able to nest this, but currently nesting isn’t working.

12.64 text.csv - CSV tables

text.csv
[Module]
Provides a function to parse/generate CSV (comma separated value) tables, including the format defined in RFC4180. You can customize the separator and quoter character to deal with variations of CSV formats.

CSV table is consisted by a series of records, separated by a newline. Each record contains number of fields, separated by a separator character (by default, a comma). A field can contain comma or newline if quoted, i.e. surrounded by double-quote characters. To include double-quote character in a quoted field, use two consecutive double-quote character. Usually, the whitespaces around the field are ignored.

Since use cases of CSV-like files vary, we provide layered API to be combined flexibly.
Chapter 12: Library modules - Utilities

Low-level API

The bottom layer of API is to convert text into list of lists and vice versa.

**make-csv-reader** separator :optional (quote-char #""")

{text.csv} Returns a procedure with one optional argument, an input port. When the procedure is called, it reads one record from the port (or, if omitted, from the current input port) and returns a list of fields. If input reaches EOF, it returns EOF.

**make-csv-writer** separator :optional newline (quote-char #"")

special-char-set

{text.csv} Returns a procedure with two arguments, output port and a list of fields. When the procedure is called, it outputs a separator-separated fields with proper escapes, to the output port. Each field value must be a string. The separator argument can be a character or a string.

You can also specify the record delimiter string by newline; for example, you can pass "\r\n" to prepare a file to be read by Windows programs.

The output of field is quoted when it contains special characters— which automatically includes characters in separator, quote-char and newline argument, plus the characters in the char-set given to special-char-set; its default is #[;\s].

Middle-level API

Occasionally, CSV files generated from spreadsheet contains superfluous rows/columns and we need to make sense of them. Here are some utilities to help them.

A typical format of such spreadsheet-generated CSV file has the following properties:

1. There’s a “header row” near the top; not necessarily the very first row, but certainly it comes before any real data. It signifies the meaning of each column of the data. There may be superfluous columns inserted just for cosmetics, and sometimes the order of columns are changed when the original spreadsheet is edited. So we need some flexibility to interpret the input data.

2. “Record rows” follow the header row. It contains actual data. There may be superfluous rows inserted just for cosmetics. Also, it’s often the case that the end of data isn’t marked clearly (you find large number of rows of empty strings, for example).

The main purpose of middle-level CSV parser is to take the output of low-level parser, which is a list of lists of strings, and find the header row, and then convert the subsequent record rows into tuples according to the header row. A tuple is just a list of strings, but ordered in the same way as the specified header spec.

**csv-rows->tuples** rows header-specs :key required-slots allow-gap?

{text.csv} Convert input rows (a list of lists of strings) to a list of tuples. A tuple is a list of slot values.

First, it looks for a header row that matches the given header-spec. Once the header row is found, parse the subsequent rows as record row according to the header and convert them to tuples. If no header is found, #f is returned.

Header-specs is a list of header spec, each of which can be either a string, a regexp, or a predicate on a string. If it’s a string, a column that exactly matches the string is picked. If it’s a regexp, a column that matches the regexp is picked. And if it’s a predicate, as you might have already guessed, a column that satisfies the predicate is picked.

The order of header-specs determines the order of columns of output tuples.

Required-slots determines if the input row is a valid record row or not. The structure of required-slots is as follows:

<required-slots> : (<spec> ...)
<spec> : <header-spec> | (<header-spec> <predicate>)

The <header-spec> compared to the elements of header-slot (by equal?) to figure out which columns to check. A single <header-spec> in <spec> means that the column shouldn’t be empty for a valid record row. If <spec> is a list of <header-spec> and <predicate>, then the value of the column corresponds to the <header-spec> is passed to <predicate> to determine if it’s a valid record row.

If required-slots is omitted or an empty list, any row with at least one non-empty column to be included in the tuple.

If allow-gap? is #t, it keeps reading rows until the end, skipping invalid rows. If allow-gap? is #f (default), it stops reading once it sees an invalid row after headers.

Let’s see an example. Suppose we have the following CSV file as data.csv. It has extra rows and columns, as is often seen in spreadsheet-exported files.

```
"Exported data",,,,,,,,,
,"Year","Country","Population","GDP","Note"
,"1958","Land of Lisp","39994","551,435,453"
,"1957","United States of Formula Translators","115333","4,343,225,434","Estimated"
,"1959","People's Republic of COBOL","82524","3,357,551,143"
,"1970","Kingdom of Pascal","3785","GDP missing"
,"1962","APL Republic","1545","342,335,151"
```

You can extract tuples of Country, Year, GDP and Population, as follows:

```scheme
(call-with-input-file "data.csv" (^p (csv-rows->tuples (generator->list (cute (make-csv-reader #
,\,) p))'("Country" "Year" "GDP" "Population"))))
```

```
⇒
(("Land of Lisp" "1958" "551,435,453" "39994")
("United States of Formula Translators" "1957" "4,343,225,434" "115333")
("People's Republic of COBOL" "1959" "3,357,551,143" "82524")
("Kingdom of Pascal" "1970" == "3785")
("APL Republic" "1962" "342,335,151")
```

Note that irrelevant rows are skipped, and columns in the results are ordered as specified in the header-specs.

Since there’s a gap (empty row) after the “Kingdom of Pascal” entry, csv-rows->tuples stops processing there by default. If you want to include “APL Republic”, you have to pass :allow-gap? #t to csv-rows->tuples.

The next example gives :required-slots option to eliminate rows with missing some of Year, Country or GDP—thus “Kingdom of Pascal” is omitted from the result, while “APL Republic” is included because of :allow-gap? argument. (It also checks Year has exactly 4 digits.)

```scheme
(call-with-input-file "data.csv" (^p (csv-rows->tuples (generator->list (cute (make-csv-reader #\,\,) p))
'("Country" "Year" "GDP" "Population")
:required-slots '(("Year" #/\d{4}\$/) "Country" "GDP")
:allow-gap? #t))
```

```
⇒
(("Land of Lisp" "1958" "551,435,453" "39994")
("United States of Formula Translators" "1957" "4,343,225,434" "115333")
("People's Republic of COBOL" "1959" "3,357,551,143" "82524")
("APL Republic" "1962" "342,335,151")
```

The following two procedures are ingredients of csv-rows->tuples:
make-csv-header-parser header-specs
{text.csv} Create a procedure that takes a row (a list of strings) and checks if it matches
the criteria specified by header-specs. (See csv-rows->tuples above about header-specs.)
If the input satisfies the spec, it returns a permuter vector that maps the tuple positions to
the input column numbers. Otherwise, it returns #f.

The permuter vector is a vector of integers, where K-th element being I means the K-th item
of the tuple should be taken from I-th column.

Let's see the example. Suppose we know that the input contains the following row as the
header row:

(define *input-row* '("" "" "Year" "Country" "Population" "GDP" "Notes"))

We want to detect that row, but we only needs Country, Year, GDP and Population columns,
in that order. So we create a header parser as follows:

(define header-parser (make-csv-header-parser '("Country" "Year" "GDP" "Population")))

Applying this header parser to the input data returns the permuter vector:

(header-parser *input-row*)
⇒ #(3 2 6 5)

It means, the first item of tuple (Country) is in the 3rd column of the input, the second item
of tuple (Year) is in the 2nd column of the input, and so on. This permuter vector can be
used to parse record rows to get tuples.

make-csv-record-parser header-slots permuter :optional required-slots
{text.csv} Create a procedure that converts one input row into a tuple.

Permuter is the vector returned by make-csv-header-parser.

See csv-rows->tuples above for header-slots and required-slots arguments.

12.65 text.diff - Calculate difference of text streams

text.diff
This module calculates the difference of two text streams or strings, using util.lcs (see
Section 12.85 [The longest common subsequence], page 1020).

diff src-a src-b :key reader eq-fn
{text.diff} Generates an "edit list" from text sources src-a and src-b.

Each of text sources, src-a and src-b, can be either an input port or a string. If it is a string,
it is converted to a string input port internally. Then, the text streams from both sources
are converted to sequences by calling reader repeatedly on them; the default of reader is
read-line, and those sequences are passed to lcs-edit-list to calculate the edit list. The
equality function eq-fn is also passed to lcs-edit-list.

An edit list is a set of commands that turn the text sequence from src-a to the one from
src-b. See the description of lcs-edit-list for the detailed explanation of the edit list.

(diff "a\nb\nc\n\nd\n" "b\n\ne\n\nf\n")
⇒
(((− 0 "a"))
  ((− 2 "c") (+ 1 "e"))
  ((+ 3 "f")))
**diff-report**  src-a src-b :key reader eq-fn writer  

A convenience procedure to take the diff of two text sources and display the result nicely. This procedure calls **lcs-fold** to calculate the difference of two text sources. The meanings of src-a, src-b, reader and eq-fn are the same as diff’s. 

Writer is a procedure that takes two arguments, the text element and a type, which is either a symbol +, a symbol -, or #f. If the text element is only in src-a, writer is called with the element and -. If the text element is only in src-b, it is called with the element and +. If the text element is in both sources, it is called with the element and #f. The default procedure of writer prints the passed text element to the current output port in unified-diff-like format:

```
(diff-report "a\nb\nc\nd\n" "b\ne\nd\nf\n")
```

displays:

- a
  b
- c
+ e
  d
+ f

**diff-report/context**  src-a src-b :key reader eq-fn writer context-size  

Produce a human-friendly difference of two text sources like diff-report, but in the “context diff” format. 

```
(diff-report/context "a\nb\nc\nd\n" "b\ne\nd\nf\n")
```

displays:

```
***************
*** 1,4 ****
- a
  b
! c
  d
--- 1,4 ----
! e
  d
+ f
```

This calls **lcs-edit-list/context** to get a context diff of lines, then format it in the same way as diff -c output. (See Section 12.85 [The longest common subsequence], page 1020, for the details of lcs-edit-list/context.)

Each “hunk” (a chunk containing difference) is preceded by a separator ***************, followed by the excerpt from src-a, then the excerpt form src-b, each preceded by the header showing the line number of that excerpts (start and end line, inclusive, 1-based).

In the excerpts, the line that’s deleted from src-a is prefixed with -, the line that’s inserted into src-b is prefixed with +, and the line changed is prefixed with !.

If the hunk only consists of either insertions or deletions, the other side of excerpt is omitted.

```
(diff-report/context "a\nb\nc\n" "a\nc\n")
```

prints:

```
***************
*** 1,3 ****
  a
- b
```
As a special case, if one of the source is empty, its header is shown as zero-th line:

```
(diff-report/context "" "a\nb\nc\n")
```

prints:

```
**************
*** 0 ****
--- 1,3 ----
+ a
+ b
+ c
```

The `reader`, `writer` and `eq-fn` keyword arguments are the same as `diff-report`. The `context-size` keyword argument specifies the maximum unchanged lines attached to each hunk to show the context.

**diff-report/unified** `src-a src-b :key reader eq-fn writer context-size`  
{text.diff} Like `diff-report/context`, but use “unified diff” format, same as `diff -u`.

```
(diff-report/unified "a\nb\nc\ndl\n" "a\nx\ndl\n")
```

prints:

```
@@ -1,4 +1,3 @@
a
-b
-c
+x
d
```

Each “hunk” begins with the header:

```
@@ -p,q +r,s @@
```

where `p` is the start line number in `src-a`, `q` is the length of the hunk in `src-a`, `r` is the start line number in `src-b`, and `s` is the length of the hunk in `src-b`.

The line-number is 1-based. If the length of the hunk is 1, it is omitted.

The lines deleted from `src-a` has `−` prefix, the ones inserted into `src-b` has `+` prefix, and unchanged lines has a space prefix.

As a special case, if either of sources is empty, both of its start index and length are shown as 0.

```
(diff-report/unified "a\nb\nc\ndl\n" "")
```

prints:

```
@@ -1,3 +0,0 @@
-a
-b
-c
```

The `reader`, `eq-fn`, `writer`, and `context-size` keyword arguments are the same as `diff-report/context`.

This is used to show the difference of test results in `test*/diff` (see Section 9.34 [Unit testing], page 521).
text.edn - EDN parsing and construction

EDN (Extensible Data Notation) is a subset of Clojure literals for data exchange. This module provides utilities to read and write EDN format. See https://github.com/edn-format/edn for the details of EDN.

<table>
<thead>
<tr>
<th>EDN</th>
<th>Gauche</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
<td>Clojure’s nil is not a symbol but a special value; since Clojure can’t have a symbol named nil, we can map it to Gauche’s symbol nil.</td>
</tr>
<tr>
<td>number</td>
<td>&lt;real&gt;</td>
<td>Integers and floating point numbers. The N and M suffixes in Clojure are ignored.</td>
</tr>
<tr>
<td>symbol</td>
<td>&lt;symbol&gt;</td>
<td>Clojure’s symbol name has some restrictions, so not all Gauche symbols map to EDN symbols. Clojure’s namespace-prefixed symbol, e.g. foo/bar simply maps to Gauche’s symbol foo/bar; we provide utility procedure to extract namespace and basename parts.</td>
</tr>
<tr>
<td>keyword</td>
<td>&lt;keyword&gt;</td>
<td>Clojure has keywords distinct from symbols. They are mapped to Gauche’s keywords (which is a subtype of symbols). Gauche’s keywords can also be symbols, but no Clojure symbols begin with : so there won’t be a conflict.</td>
</tr>
<tr>
<td>list</td>
<td>&lt;list&gt;</td>
<td>Clojure lists are Gauche lists. Note that Clojure doesn’t allow improper lists.</td>
</tr>
<tr>
<td>vector</td>
<td>&lt;vector&gt;</td>
<td>Clojure vectors are Gauche vectors.</td>
</tr>
<tr>
<td>map</td>
<td>&lt;hash-table&gt;</td>
<td>Clojure’s map becomes Gauche’s hashtable with edn-comparator for hashing and comparison.</td>
</tr>
<tr>
<td>set</td>
<td>&lt;set&gt;</td>
<td>Clojure’s set becomes Gauche’s set with edn-comparator for comparison. See Section 10.3.5 [R7RS sets], page 606, for interface of sets.</td>
</tr>
<tr>
<td>tagged object</td>
<td>&lt;edn-object&gt;</td>
<td>Tagged objects are mapped to &lt;edn-object&gt; by default. You can customize the parser/writer to map tagged objects with a specific tag to a specific Gauche objects.</td>
</tr>
</tbody>
</table>

Parsing

<edn-parse-error> [Condition Type]
When the parser encounters an error, this condition is thrown. Inherits <error>.

parse-edn :optional iport [Function]
{text.edn} Read one EDN representation from the given input port, and returns Gauche object created from it. If iport is omitted, current input port is assumed.

When the parser encounters unparsable sequence, it raises <edn-parse-error>.

Note that iport may be read ahead for characters. Suppose the input consists of abc{:a b}, i.e. a symbol immediately followed by a map. The parser need to read {} to know the end of the symbol. The read-ahead brace isn’t pushed back to the iport. So it would be a problem if you keep reading more EDN subsequently. Use parse-edn* if you want to read multiple objects.
parse-edn*: :optional iport

{text.edn} Read EDN representations repeatedly from the given input port and returns a list of them. If iport is omitted, current input port is assumed.

When the parser encounters unparsable sequence, it raises <edn-parse-error>.

parse-edn-string str

{text.edn} A convenience procedure to parse EDN representation in a string str, and returns the read object.

When the parser encounters unparsable sequence, it raises <edn-parse-error>.

( parse-edn-string "[1 2 (3 4) {:a 5}]"
⇒ #(1 2 (3 4) #<hash-table general 0x1f05780>)

Constructing

construct-edn obj :optional oport

{text.edn} Write out an EDN representation of object obj to the output port oport. If oport is omitted, current output port is assumed.

If obj contains an object that doesn’t have a defined EDN representation, a generic function edn-write is called on it. See Customization heading below. If no method is defined for the object, an error is signaled.

construct-edn-string obj

{text.edn} Returns an EDN representation of obj in a string.

If obj contains an object that doesn’t have a defined EDN representation, a generic function edn-write is called on it. See Customization heading below. If no method is defined for the object, an error is signaled.

(construct-edn-string #'(1 2 "abc") ⇒ ":[1 2 "abc"]"

Utilities

edn-equal? a b

{text.edn} Test equality of two objects that are read from EDN representation.

edn-comparator

{text.edn} A comparator that uses edn-equal? for the equality predicate. Corresponding has function is also included. EDN maps and sets become Gauche hash-tables and sets with this comparator.

edn-map key value ... 
edn-set item ... 

{text.edn} Convenience procedures to create hash-tables and sets compatible for EDN.

<edn-object>

{text.edn} EDN tagged object becomes an instance of this class by default. The instance has the following slots, both are immutable:

tag
Object’s tag. A symbol.

payload
Object’s payload. Can be any object that can be representable in EDN.

For example, when you read #myobject {:a 1 :b 2}, the tag is myobject and the payload is a hashtable containing mapping {:a 1 :b 2}.
make-edn-object  tag payload
{text.edn} Returns a new <edn-object> instance. Note: Arguments are not checked. It’s
caller’s responsibility to pass valid arguments to guarantee it’s serializable as EDN.

edn-object?  obj
{text.edn} Returns #t if obj is an instance of <edn-object>.

edn-object-tag  edn-object
edn-object-payload  edn-object
{text.edn} Returns the tag and the payload of edn-object, respectively.

edn-symbol-prefix  symbol
edn-symbol-basename  symbol
{text.edn} Return prefix and basename part of the symbol, respectively.

(edn-symbol-prefix 'foo/bar) ⇒ foo
(edn-symbol-basename 'foo/bar) ⇒ bar
(edn-symbol-prefix 'bar) ⇒ #f
(edn-symbol-basename 'bar) ⇒ bar

edn-valid-symbol-name?  str
{text.edn} Returns #t if a string str can be a valid Clojure symbol name. It may have
namespace prefix.

Customization
You can map EDN tagged objects to other Gauche objects.

register-edn-object-handler!  tag handler
{text.edn} Tag is a symbol, and handler is #f or a procedure that takes a tag symbol and
a payload object.

Tag must have a name valid as Clojure symbol, or an error is signaled.

After the parser reads a tagged object with a symbol tag and payload, it calls handler, and
the returned object becomes the result of the parser, instead of <edn-object>. Registering
#f removes the previously registered handler.

This procedure is thread-safe.

The following example makes EDN #u8vector[1 2 3 4] to be read as #u8(1 2 3 4):

(register-edn-object-handler! 'u8vector
("^[tag vec] (vector->u8vector vec))

edn-object-handler  tag
{text.edn} Returns a handler registered with a symbol tag. If a handler is not registered
for tag, #f is returned. This procedure is thread-safe.

edn-write  obj
{text.edn} Write EDN representation of obj to the current output port. The construct-edn
procedure calls this internally.

To write out a Gauche object as EDN tagged object, define a method to this generic function.
In the method you can call edn-write recursively to write out components of the object.

The following example writes #u8(1 2 3 4) as EDN #u8vector[1 2 3 4]:

(define-method edn-write ((x <u8vector>))
(display "#u8vector")
(edn-write (u8vector->vector x)))
12.67 text.external-editor - Running external editor

text.external-editor

A convenience module to invoke external editor program. This is mainly to be used from REPL, but can be useful for other purposes.

ed path-or-proc :key editor load-after

{text.external-editor} Starts an external editor. The editor program path is determined in the following order:

- The editor keyword argument, if it’s a string.
- The value of *editor* in the user module, if defined.
- The value of the environment variable GAUCHE_EDITOR.
- The value of the environment variable EDITOR.
- If none works, the behavior depends on the value of editor keyword argument:
  
  #f Just returns #f.
  
  error Throws an error.
  
  ask Asks the user. (default)
  
  message Prints message and returns #f.

Returns #t for normal termination.

The editor program may be called in one of the following way:

  EDITOR filename
  EDITOR +lineno filename

The latter expects to locate the cursor on the specified line number in the filename.

The file to open is determined by calling the generic function ed-pick-file on the argument path-or-proc. It should return (filename lineno), or #f to indicate that it couldn’t determine the file to edit. Methods for <string> and <procedure> are already defined; see the entry of ed-pick-file below.

The load-after argument controls whether the file is loaded into the process after the file is modified. It must be one of the following values:

  #t Load the file once editor exits and the file is modified.
  
  #f Do not load the file.
  
  ask Ask the user if the file should be loaded or not if the file is modified. (default)

NB: Common Lisp’s ed can be invoked without argument. For our usage, though, that feature doesn’t seem too useful—it’s more likely that repl IS inside an editor so we only need to open a specific file in a buffer. Emacsclient requires filename, so it further complicates things. For now we just require one argument.

ed-string string :key editor

{text.external-editor} Invoke an external editor (as described in ed) on a temporary file whose content is string. Once editor exits, returns the edited content as a string.

ed-pick-file obj

{text.external-editor} Determine what to edit. It must return a list of filename and line number, or #f if it can’t find appropriate file to edit.

Methods for strings, procedures and <top> are already defined. If obj is a string, it is taken as a filename. If obj is a procedure and its source location is available, the source file and the location is returned. Otherwise, #f is returned.
12.68 text.gap-buffer - Gap buffer

**text.gap-buffer**

This module provides a gap buffer, a data structure useful for editable text.

A gap buffer is a vector of characters with a “cursor”, where you can insert or delete a character in O(1) time. Most of the editing API operate on the current cursor position.

**make-gap-buffer :key initial-capacity**

{text.gap-buffer} Creates a fresh gap buffer and returns it. The initial content is empty.

The keyword argument initial-capacity must be a positive exact integer if given, and specifies the initial capacity of the buffer. The buffer is automatically extended if necessary, so the argument is only a hint; if you know you’ll insert a long string, for example, you can create a buffer that’s large enough to hold it to avoid reallocation overhead.

**string->gap-buffer string :optional pos whence start end**

{text.gap-buffer} Creates a gap buffer that contains the given string, and returns it.

The initial cursor position is set at the end of the string by default. The pos and whence argument can set the initial cursor position. The whence argument must be either a symbol beginning or a symbol end, and pos must be an exact integer offset from the whence. For example, pos = 10 and whence = beginning sets the cursor at the 10th character of the string from the beginning, and pos = -1 and whence = end sets the cursor position at one character before the end of the string. If the combination of pos and whence points the outside of the string, an error is signaled.

The optional start and end trims the input string before creating the gap buffer.

**gap-buffer? obj**

{text.gap-buffer} Returns #t iff obj is a gap buffer.

**gap-buffer-copy gbuf**

{text.gap-buffer} Returns a fresh copy of a gap buffer gbuf, with the same content and cursor position.

**gap-buffer->string gbuf :optional start end**

**gap-buffer->generator gbuf :optional start end**

{text.gap-buffer} Retrieve the content of a gap buffer gbuf as a string or a generator of characters, respectively. The optional start and end arguments are nonnegative exact integers of character index in the content of the buffer, and limits the output within the specified range.

If you modify the content of gap buffer before the returned generator is exhausted, consistent behavior isn’t guaranteed.

**gap-buffer-pos gbuf**

{text.gap-buffer} Returns the current cursor position of gbuf, in a nonnegative exact integer.

**gap-buffer-pos-at-end? gbuf pos**

{text.gap-buffer} Returns #t if pos, a position in nonnegative integer, is at the end of gbuf, #f otherwise.

**gap-buffer-capacity gbuf**

{text.gap-buffer} Returns the currently allocated storage size of gbuf, in character count.

If you insert more characters into gbuf, the storage is automatically expanded.
gap-buffer-content-length gbuf
   {text.gap-buffer} Returns the length of current content of gbuf, in the number of characters.

gap-buffer-gap-at? gbuf whence
   {text.gap-buffer} The whence argument must be either a symbol beginning or a symbol end. Returns #t iff the current cursor position of gbuf is at the beginning or at the end, respectively.

gap-buffer-ref gbuf index :optional fallback
   {text.gap-buffer} Returns index-th character of gbuf. If index is out of range, fallback is returned if given, or an error is raised.

gap-buffer-set! gbuf index char
   {text.gap-buffer} Replaces index-th character of gbuf with char. If index is out of range, an error is signaled.

gap-buffer-move! gbuf pos :optional whence
   {text.gap-buffer} Moves the cursor position of gbuf to a position pos, which must be an exact integer. The position pos is relative to whence, which must be either one of the symbols beginning, current or end. If omitted, beginning is assumed.

gap-buffer-insert! gbuf content
   {text.gap-buffer} Inserts content, which must be a character or a string, at the current cursor position of gbuf. The current cursor position is moved to the end of the inserted content.

gap-buffer-delete! gbuf size
   {text.gap-buffer} Deletes size characters from the current cursor position of gbuf. The size argument must be a nonnegative exact integer. It is an error if size overruns.

gap-buffer-clear! gbuf
   {text.gap-buffer} Makes gbuf empty.

gap-buffer-replace! gbuf size content
   {text.gap-buffer} This is a combination of (gap-buffer-delete! gbuf size) and (gap-buffer-insert! gbuf content). The size argument must be a nonnegative exact integer, and the content argument must be either a string or a character.

gap-buffer-contains gbuf str :optional gstart gend sstart send
   {text.gap-buffer} Search a string str in a gap buffer gbuf. If found, returns the integer index of the beginning of str in gbuf. If not found, returns #f.
   The optional arguments gstart and gend restricts the range of gbuf to be searched, by integer indexes. The default is the entire buffer. You can pass #f to indicate to use the default value.
   The optional arguments sstart and send restricts the span of str to be searched, by integer indexes. You can pass #f to indicate to use the default value.

gap-buffer-looking-at? gbuf str :optional point
   {text.gap-buffer} Returns #t iff the content immediately following point matches a string str. If point is omitted or #f, the current cursor position is used.

gap-buffer-edit! gbuf edit-command
   {text.gap-buffer} This is a convenience procedure to “replay” editing of the gap buffer gbuf. The edit-command argument must be either one of the following:
   (i pos string)    Insert string at a position pos.
(d pos length)
Delete length characters from a position pos.

(c pos length string)
Change: Deletes length characters from a position pos, then insert string.
Here, pos must be a nonnegative exact integer specifying the position in the gap buffer.
It returns another edit-command that undo the change.
One of the applications of this procedure is to handle undo/redo list.

(define buf (string->gap-buffer "Hello, World!"))

(gap-buffer-edit! buf '(d 0 5))
⇒ (i 0 "Hello")

(gap-buffer-edit! buf '(i 0 "Konnichiwa"))
⇒ (d 0 10)

(gap-buffer->string buf)
⇒ "Aloha, World!"

;; Giving the returned values of gap-buffer-edit! in reverse order,
;; you can undo the changes:

(for-each (cut gap-buffer-edit! buf <>) '((d 0 10) (i 0 "Hello")))
(gap-buffer->string buf)
⇒ "Hello, World!"

12.69 text.gettext - Localized messages

text.gettext [Module]
This module provides utilities to deal with localized messages. The API is compatible to
GNU’s gettext, and the messages are read from *.po and *.mo files, so that you can use
the GNU gettext toolchain to prepare localized messages. However, the code is written from
scratch by Alex Shinn and doesn’t depend on GNU’s gettext library.

This implementation extends GNU’s gettext API in the following ways:
• It can read from multiple message files in cascaded way, allowing applications to share a
  part of message files.
• It supports multiple locale/domain simultaneously.

SRFI-29 (see Section 11.9 [Localization], page 707) provides another means of message lo-
calization. A portable program may wish to use SRFI-29, but generally text.gettext is
recommended in Gauche scripts because of its flexibility and compatibility to existing mes-
sage files.

Gettext-compatible API

textdomain domain-name \optional locale dirs cdrs cdir cached?\ lookup-cached?\ {text.gettext} Sets up the default domain and other parameters for the application. The
setting affects to the following gettext call.
Domain is a string or list of strings specifying the domain (name of .mo or .po files) as in C
gettext. You can pass #f as domain-name just to get the default domain accessor procedure.
You can also pass multiple domains to domain-name.
Chapter 12: Library modules - Utilities

Locale is a string or list of strings in the standard Unix format of LANG[_REGION][_ENCODING]. You can also pass a list of locales to specify fallbacks.

Dirs is the search path of directories which should hold the LOCALE/CDIR/ directories which contain the actual message catalogs. This is always appended with the system default, e.g. "/usr/share/locale", and may also inherit from the GETTEXT_PATH colon-delimited environment variable.

Cdir is the category directory, defaulting to either the LC_CATEGORY environment variable or the appropriate system default (e.g. LC_MESSAGES). You generally won’t need this.

Cached? means to cache individual messages, and defaults to #t.

Lookup-cached? means to cache the lookup dispatch generated by these parameters, and defaults to #t.

Textdomain just passes these parameters to the internal make-gettext, and binds the result to the global dispatch used by gettext. You may build these closures manually for convenience in using multiple separate domains or locales at once (useful for server environments). See the description of make-gettext below.

Textdomain returns an accessor procedure which packages information of the domain. See make-gettext below for the details.

gettext msg-id
{text.gettext} Returns a translated message of msg-id. If there’s no translated message, msg-id itself is returned.

ngettext msg-id :optional msg-id2 num
{text.gettext} Similar to gettext, but it can be used to handle plural forms. Pass a singular form to msg-id, and plural form to msg-id2. The num argument is used to determine the plural form. If no message catalog is found, msg-id is returned when num is 1, and msg-id2 otherwise.

bindtextdomain domain dirs
{text.gettext} Sets the search path of domain domain to dirs, which may be just a single directory name or a list of directory names.

dgettext domain msg-id
dcgettext domain msg-id locale
{text.gettext} Returns a translated message of msg-id in domain. Dcgettext takes locale as well.

Low-level flexible API

The following procedure is more flexible interface, on top of which the gettext-compatible APIs are written.

make-gettext :optional domain locale dirs gettext-cached?
lookup-cached?
{text.gettext} Creates and returns an accessor procedure, which encapsulates methods to retrieve localized messages.

The meaning of arguments are the same as textdomain above. Indeed, textdomain just calls make-gettext, and later it binds the result to the global parameter. If you wish to have
multiple independent domains within a single program, you can call make-gettext directly
and manage the created accessor procedure by yourself.

(define my-gettext (make-gettext "myapp"))
(define _ (my-gettext 'getter))
(_ "Hello, World!")

12.70 text.html-lite - Simple HTML document construction

text.html-lite [Module]
Provides procedures to construct an HTML document easily. For example, you can construct
an HTML table by the following code:

(html:table
  (html:tr (html:th "Item No") (html:th "Quantity"))
  (html:tr (html:td 1) (html:td 120))
  (html:tr (html:td 2) (html:td 30))
  (html:tr (html:td 3) (html:td 215)))

See the description of html:element below for details.
This module does little check for the constructed html documents, such as whether the
attributes are valid, and whether the content of the element matches DTD. It does not
provide a feature to parse the html document neither. Hence the name ‘lite’.

html-escape [Function]
html-escape-string string [Function]
{text.html-lite} Escapes the “unsafe” characters in HTML. html-escape reads input
string from the current input port and writes the result to the current output port.
html-escape-string takes the input from string and returns the result in a string.

html-doctype :key type [Function]
{text.html-lite} Returns a doctype declaration for an HTML document. type can be
either one of the followings (default is :html-4.01-strict).

:html-4.01-strict, :html-4.01, :strict
  HTML 4.01 Strict DTD
:html-4.01-transitional, :transitional
  HTML 4.01 Transitional DTD
:html-4.01-frameset, :frameset
  HTML 4.01 Frameset DTD
:xhtml-1.0-strict, :xhtml-1.0
  XHTML 1.0 Strict DTD
:xhtml-1.0-transitional
  XHTML 1.0 Transitional DTD
:xhtml-1.0-frameset
  XHTML 1.0 Frameset DTD
:xhtml-1.1
  XHTML 1.1 DTD

html:element args ... [Function]
{text.html-lite} Construct an HTML element element. Right now, the following elements
are provided. (The elements defined in HTML 4.01 DTD, http://www.w3.org/TR/html4/
sgml/dtd.html).

a   abbr   acronym   address   area   b
The result of these functions is a tree of text segments, which can be written out to a port by write-tree or can be converted to a string by tree->string (see Section 12.80 [Lazy text construction], page 1014).

You can specify attributes of the element by using a keyword-value notation before the actual content.

\[(\text{tree->string } (\text{html:a :href "http://foo/bar" "fooban"}))\]
\[⇒ "<a href="http://foo/bar">fooban</a>"\]

\[(\text{tree->string } (\text{html:table :width "100\%" :cellpadding 0 "content here"}))\]
\[⇒ "<table width="100\%" cellpadding="0">content here</table>"\]

The boolean value given to the attribute has a special meaning. If #t is given, the attribute is rendered without a value. If #f is given, the attribute is not rendered.

\[(\text{tree->string } (\text{html:table :border #t}))\]
\[⇒ "<table border></table>"\]

\[(\text{tree->string } (\text{html:table :border #f}))\]
\[⇒ "<table></table>"\]

Special characters in attribute values are escaped by the function, but the ones in the content are not. It is caller’s responsibility to escape them.

The functions signal an error if a content is given to the HTML element that doesn’t take a content. They do not check if the given attribute is valid, neither if the given content is valid for the element.

Note: You might have noticed that these procedures insert a newline before > of the closing tag. That is, the rendered HTML would look like this:

\[
<\text{table}><\text{tr}><\text{td}>foo</\text{td}>\text{bar}</\text{td>}
</\text{tr}>
</\text{table}>
\]

We intentionally avoid inserting newlines after the closing tag, since it depends on the surrounding context whether the newline is significant or not. We may be able to insert newlines after the elements directly below a <head> element, for example, but we cannot in a <p> element, without affecting the content.
There are three possible solutions: (1) not to insert newlines at all, (2) to insert newlines within tags, and (3) to insert newlines only at the safe position. The first one creates one long line of HTML, and although it is still valid HTML, it is inconvenient to handle it with line-oriented tools. The third one requires the rendering routine to be aware of DTD. So we took the second approach.

12.71 text.info - Accessing info document

text.info

This module provides some basic functionalities to extract information from info files, a documentation format defined by texinfo system (https://www.gnu.org/software/texinfo/). It is used in REPL toplevel commands info and doc (see Section 3.2.1 [Working in REPL], page 24). It focuses mainly to handle Gauche documents, so it may not handle all features used in texinfo.

<info-document>

{text.info} An object for opened info file(s), returned by open-info-document. All slots are private.

<info-node>

{text.info} An object for each info node. It keeps the node's content, as well as links to neighboring nodes. Its instance should be considered immutable object.

A node can be obtained from an <info-document> object by info-get-node.

name [Instance Variable of <info-node>]

A string name of the info node. Every info document has at least "Top" node.

next [Instance Variable of <info-node>]

The string name of the next node. You can visit all nodes in the document following the next link. It may be #f if the node is the last one of the document.

prev [Instance Variable of <info-node>]

The string name of the previous node. It’s a reverse link of the next node. The previous node of "Top" is "(dir)" by convention, which is a special node that lists all info documents installed on the system (it is not included in the info document itself.)

up [Instance Variable of <info-node>]

If the node is a subnode, this contains the name of the upper node (a chapter name for sections, a section name for subsections, etc.) It is #f for "Top" node.

file [Instance Variable of <info-node>]

An <info-document> object of the info document this node is contained.

content [Instance Variable of <info-node>]

A string content of the node.

open-info-document info-file-path

{text.info} Open the named info document and returns an <info-document> instance. The argument is a pathname to a master info file, usually with .info suffix. Depending on the installation, info files may be compressed with gzip or bzip2, having extensions .gz or .bz2 after .info; you don’t need to specify those extensions. The procedure searches possible compression options.

A large info document may be splitted into multiple info files, each with a suffix .info-1, .info-2 etc. Those are automatically read, so you only need to give the path to the master file.

If it can’t find the named document, or can’t recognize its format, an error is thrown.
info-get-node info-doc node-name
{text.info} The info-doc argument is an <info-document> instance. Returns an <info-node> instance with the string name node-name, or #f if no node with the given name is found.

info-index-add! info-doc index-node-name :optional key-modifier
{text.info} Read the node named index-node-name in the info document info-doc, and adds its menu entries into the index table.

It is particularly useful to give the index page of the info doc, so that you’ll be able to lookup particular term (e.g. function name) quickly. Typical info document has a node called Index. Large documents, such as programming language reference (Gauche Users’ Reference is one of them) may have multiple index pages for different categories. You can call this procedure multiple times to add more than one index pages.

The optional key-modifier argument is a procedure applied to the entry-name to obtain a key in the index table. Sometimes the index uses different entry name from the actual name; e.g. Gauche’s class index lists class names without surrounding < and >, since using the actual name makes all class names being listed below < subheading, which isn’t very useful. You can pass (^e #"<"|e|>") as key-modifier to recover the actual class name to be used as the key.

If there are more than one entries per key, both are saved in the index table. See info-lookup-index below.

info-index-ref info-doc key
{text.info} Lookup index with the given key. Returns a list of (<node-name> <line-number>).

info-index-keys info-doc
{text.info} Returns a list of keys (strings) in the index.

info-index->alist info-doc
{text.info} Returns the content of the index as ((key (node line) ...) ...).

info-parse-menu info-node
{text.info} This is used in info-index-add!, but it is generally useful to parse a menu in an info node. It searches the * Menu: line, then gathers menu entry names and associated node names. Some menus such as index page also lists line numbers for each entry.

Returns a list of menu entries. Each menu entry is the following form:

(<entry-name> <node-name> [line-number])

info-extract-definition info-node start-line
{text.info} Extract one definition from the node’s content. Assumes the definition begins from the specified line; then we go forward to find the end of the definition. The end of definition is when we see the end of content, or we see a line begins with less than or equal to 3 whitespaces. (Except the 'defunx'-type multi entry).

Returns a string describing the definition. If no definition is found, an empty string is returned.

A typical idiom is to add index node using info-index-add!, and get the node name and line number of the definition of a specific item with info-index-ref, then then use this procedure to extract the definition.
12.72 text.multicolumn - Multicolumn formatting

This module provides the means to format a list of words into a multicolumn text. Think of the output of `ls` command.

```
(display-multicolumn
 (string-tokenize "The quick brown fox \jumps over the lazy dog"))
⇒
prints
The  fox  the
quick  jumps  lazy
brown  over  dog
```

Various options are provided to customize the output.

**display-multicolumn strs :key width minimum-width max-columns order**

{text.multicolumn} Display items in a list of strings `strs` in multicolumn format, to the current output port. The items shouldn't contain newline or tab characters; the procedure doesn't treat them specially, so the output would be disturbed.

The width of a column is chosen to contain the longest string in `strs` or `minimum-width`, whichever greater. The default of `minimum-width` is 8.

Then as many columns are displayed as fit in `width` characters, whose default is 80. However, the number of columns won't exceed `max-columns` if it is given and not `#f`.

The `order` argument must be either a symbol `column` or `row`. If it is `column` (default), the items are sorted column-first; that is, it is filled in the first column from top to down, then the second column from top to down, and so on. If it is `row`, the items are sorted row-first; it is filled in the first row from left to right, and then second row, and so on.

```
(layout-multicolumn
 (string-tokenize "The quick brown fox \jumps over the lazy dog"))
⇒
(("The"  "fox"  "the")
 ("quick"  "jumps"  "lazy")
 ("brown"  "over"  "dog"))
```

NB: The actual structure of each text tree may be changed, but passing it to `tree->string` or `write-tree` produces the line.

12.73 text.pager - Display with pager

A convenience module to present long text nicely to the user.

**pager-program**

{text.pager} A parameter containing a list of pager program and its arguments (e.g. `("/usr/bin/less" "-M")`).
The program must take input from the standard input. The default value is taken from the environment variable $\text{PAGER}$ if set, or either $\text{less}$ or $\text{more}$ if those programs are available on the system. If none is found, $\text{#f}$ is set.

**display/pager string**  
{\text{text.pager}} Display string by a pager subprocess specified by the parameter $\text{pager-program}$. The procedure returns when the subprocess exits.  
If the terminal is not suitable for control (e.g., $\text{TERM}$ is $\text{dumb}$ or $\text{emacs}$), string is simply displayed without a pager.  
If you’re running MinGW version of $\text{gosh}$ on mintty, calling subprocess pager doesn’t work well, so the procedure emulate the pager behavior ($\text{pager-program}$ is ignored).

**with-output-to-pager thunk**  
{\text{text.pager}} Call thunk with its current output port to a string buffer, then show the buffered output using $\text{display/pager}$.

### 12.74 text.parse - Parsing input stream

**text.parse**  
A collection of utilities that does simple parsing from the input port. The API is inspired, and compatible with Oleg Kiselyov’s input parsing library (http://okmij.org/ftp/Scheme/parsing.html). His library is used in lots of other libraries, notably, a full-Scheme XML parser/generator SSAX (http://okmij.org/ftp/Scheme/xml.html).  
You can use this module in place of his $\text{input-parse.scm}$ and $\text{look-for-str.scm}$.  
I reimplemented the functions to be efficient on Gauche. Especially, usage of $\text{string-set!}$ is totally avoided. I extended the interface a bit so that they can deal with character sets and predicates, as well as a list of characters.  
These functions work sequentially on the given input port, that is, they read from the port as much as they need, without buffering extra characters.

**find-string-from-port? str in-port :optional max-no-chars**  
{\text{text.parse}} Looks for a string str from the input port in-port. The optional argument max-no-chars limits the maximum number of characters to be read from the port; if omitted, the search span is until EOF.  
If str is found, this function returns the number of characters it has read. The next read from in-port returns the next char of str. If str is not found, it returns $\text{#f}$.  
Note: Although this procedure has ‘?’ in its name, it may return non-boolean value, contrary to the Scheme convention.

**peek-next-char :optional port**  
{\text{text.parse}} Discards the current character and peeks the next character from port. Useful to look ahead one character. If port is omitted, the current input port is used.  

In the following functions, char-list refers to one of the followings:  
- A character set.  
- A list of characters, character sets and/or symbol $\text{*eof*}$.

That denotes a set of characters. If a symbol $\text{*eof*}$ is included, the EOF condition is also included. Without $\text{*eof*}$, the EOF condition is regarded as an error.

**assert-curr-char char-list string :optional port**  
{\text{text.parse}} Reads a character from port. If it is included in char-list, returns the character. Otherwise, signals an error with a message containing string. If port is omitted, the current input port is used.
skip-until char-list/number :optional port
{text.parse} char-list/number is either a char-list or a number. If it is a number; it reads that many characters and returns #f. If the input is not long enough, an error is signaled. If char-list/number is a char-list, it reads from port until it sees a character that belongs to the char-list. Then the character is returned. If port is omitted, the current input port is used.

skip-while char-list :optional port
{text.parse} Reads from port until it sees a character that does not belong to char-list. The character remains in the stream. If it reaches EOF, an EOF is returned. If port is omitted, the current input port is used. This example skips whitespaces from input. Next read from port returns the first non-whitespace character.

(next-token-of char-list/pred :optional port)
{text.parse} Reads and collects the characters as far as it belongs to char-list/pred, then returns them as a string. The first character that doesn’t belong to char-list/pred remains on the port.

next-token prefix-char-list break-char-list :optional comment port
{text.parse} Skips any number of characters in prefix-char-list, then collects the characters until it sees break-char-list. The collected characters are returned as a string. The break character remains in the port. If the function encounters EOF and *eof* is not included in break-char-list, an error is signaled with comment is included in the message.

Reads and collects the characters as far as it belongs to char-list/pred, then returns them as a string. The first character that doesn’t belong to char-list/pred remains on the port.

char-list/pred may be a char-list or a predicate that takes a character. If it is a predicate, each character is passed to it, and the character is regarded to “belong to” char-list/pred when it returns a true value.

read-string n :optional port
{text.parse} This is like built-in read-string (see Section 6.21.7.1 [Reading data], page 270), except that this returns "" when the input already reached EOF. Provided for the compatibility for the code that depends Oleg’s library.

12.75 text.progress - Showing progress on text terminals

text.progress
This module provides a utility to report a progress of processing on a text terminal, using characters to display bar chart.

The generic format of a progress bar consists of a single line of text, which is splitted into several parts; a header, which displays the title; followed by a bar, a numeric part, and a time part, as shown in the following example (only the line beginning with “foo” is actually displayed).

```
<-header->  <--------bar----------->  <-num-><-time->  <---info----->
foo  |################|123/211 01:21 ETA compiling...
```

Various things like the character used in the bar chart or the format of the numeric progress can be configured.

Internally a progress bar maintains two numbers, the maximum (goal) value and the current value. The bar shows the proportion of the current value relative to the maximum value.
The numeric progress shows the current value over the maximum value by default, but you can configure it to show only the current value or percentage, for example.

A progress bar also has two states, “in progress” and “finished”. When it is in progress, every time the text is displayed it is followed by \texttt{\#return}, so that the next display overwrites the bar, and the time part shows ETA (estimated time of arrival). Once it becomes finished, the last line of text is displayed with \texttt{\#newline}, and the time part shows the actual time it took to finish.

This module provides only one procedure, \texttt{make-text-progress-bar}, which packages the progress bar feature in a closure and returns it.

\textbf{make-text-progress-bar} \texttt{:key header header-width bar-char bar-width} \hspace{1em} [Function]
\texttt{num-width num-format time-width info info-width separator-char max-value}
\texttt{port}
\texttt{\{text.progress\}} Returns a procedure that packages operations on the progress bar. The procedure can be called with a symbol indicating an operation, and an optional numeric argument.

\texttt{proc 'show}
Redisplays the progress bar. All other operations implies redisplay, so you don’t need to use this unless you have a specific reason to redisplay the current state.

\texttt{proc 'set value}
Sets the current value to \texttt{value}, then redisplays the progress bar. If \texttt{value} exceeds the max value, it is clipped by the max value.

\texttt{proc 'inc value}
Increments the current value by \texttt{value}, then redisplays the progress bar. If the current value exceeds the max value, it is clipped by the max value.

\texttt{proc 'finish}
Puts the progress bar to the “finished” state, then redisplays it. The time part shows the total elapsed time, and the line is terminated by \texttt{\#newline} so that it won’t be clobbered. Once a progress bar becomes “finished”, there’s no way to put it back “in progress”.

\texttt{proc 'set-info text}
Changes the text displayed in the “info” part. To use the info part, you have to give a positive value to \texttt{info-width} keyword argument of \texttt{make-text-progress-bar}.

\texttt{proc 'set-header text}
Changes the text displayed in the “header’ area.

The keyword arguments are used to customize the display:

\texttt{header} The text to be displayed in the header part. This can be changed later, by sending \texttt{set-header} message to the created progress bar.

\texttt{header-width} The width of the header part, in number of characters. The header text is displayed left-aligned in the part. If the header text is longer than the width, the excess characters are omitted. The default is 14.

\texttt{bar-char} A character used to draw a bar chart. The default is \texttt{###}.

\texttt{bar-width} The width of the bar chart part, in number of characters. The default is 40.
**num-width**
The width of the numeric part, in number of characters. The default is 9. Setting this to 0 hides the numeric part.

**num-format**
A procedure to format the numeric part. Two arguments are passed: the current value and the maximum value. It must return a string. The default is the following procedure.

\[
\text{(lambda (cur max) \( \text{format "~d/\~d" cur max} \))}
\]

**time-width**
The width of the time part, in number of characters. The default is 7. Setting this to 0 hides the time part.

**info**
The text to be displayed in the info part. This text can be changed later by sending `set-info` message to the created progress bar. Note that you have to give a positive number to `info-width` keyword argument to enable the info part.

**info-width**
The width of the info part. The default value is zero, which means the info part is not displayed.

**separator-char**
A character put around the bar part. Default is `#\|`. You can pass `#\f` not to display the separators.

**max-value**
The maximum value of the progress bar. Must be a positive real number. Default is 100.

**port**
An output port to which the progress bar is displayed. The default value is the current output port when `make-text-progress-bar` is called.

Here’s a simple example, using customized numeric part:

```scheme
(use text.progress)

(define (main args)
  (define (num-format cur max)
    (format "~d/\~d(~3d\%)") cur max
    (round->exact (/ (* cur 100) max)))))

  (let ((p (make-text-progress-bar :header "Example"
    :header-width 10
    :bar-char #\o
    :num-format num-format
    :num-width 13
    :max-value 256)))
    (do ((i 0 (+ i 1)))
      ((= i 256) (p 'finish)
        (p 'inc 1)
        (sys-select #f #f #f 50000)))))

12.76 text.segmented-match - Segmented string matching

**text.segmented-match**
Sometimes you want to search for a group of words concatenated with a delimiter, using only the prefixes of each words. For example: you want to search `call-with-current-`
Chapter 12: Library modules - Utilities

continuation using c-w-c-c. One of its popular applications is symbol completion, which
Gauche provides on REPL (see Section 3.2.2 [Input editing], page 29).
This module provides basic parts to perform such searching.

**segmented-prefix?** pattern word :optional separator  
{text.segmented-match} The pattern and word arguments must be a string, and separator
must be either a character, a character set, a regexp, a predicate taking a character, or a
string. When separator is omitted, #\- is assumed.
First, pattern and word are splitted into segments by separator, using string-split (see
Section 6.11.9 [String utilities], page 178). If every segments from pattern consist a prefix of
the corresponding segments from word, it returns #t. It is allowed for word to have more
segments than pattern.

```
(segmented-match? "a-b-c" "alpha-bravo-charlie") ⇒ #t
(segmented-match? "a-b" "alpha-bravo-charlie") ⇒ #t
(segmented-match? "al-br-char" "alpha-bravo-charlie") ⇒ #t
(segmented-match? "a-b" "alpha-delta-charlie") ⇒ #t
```

**make-segmented-prefix-matcher** pattern separator  
{text.segmented-match} This is a subfunction of segmented-prefix?.
Returns a procedure that takes a string word, and sees if each segment of pattern is a prefix
of corresponding segment of word. If any of the pattern’s segment is not a prefix of the
corresponding word’s segment, #f is returned. Otherwise, it returns a true value: If the
number of word’s segments is the same as the pattern’s, #t is retured; if word has more
segments, the remaining part of the word (excluding leading separator) is returned.

```
((make-segmented-prefix-matcher "a-b-c" #\-) "alpha-bravo-charlie")
⇒ #t
((make-segmented-prefix-matcher "a-b" #\-) "alpha-bravo-charlie-delta")
⇒ "charlie-delta"
((make-segmented-prefix-matcher "a-b" #\-) "alpha-charlie-delta")
⇒ #f
```

This preprocesses pattern, so if you want to match many words against a single pattern, it is
faster to get predicates with this procedure than applying segmented-prefix? repeatedly.

12.77 text.sql - SQL parsing and construction

text.sql  
This module provides a utility to parse and construct SQL statement.
It is currently under development, and we only have a tokenization routine. The plan is to
define S-expression syntax of SQL and provides a routine to translate one form to the other.
Note: If you’re looking for a routine to escape strings to be safe in SQL, see dbi-escape-sql
in Section 12.27.1 [DBI user API], page 865.

**sql-tokenize** sql-string  
{text.sql} Tokenize a SQL statement sql-string. The return value is a list of tokens, where
each token is represented by one of the following forms.

```
<symbol> Special delimiter. One of the followings: + - * / < = > <= => >= ||
<character> Special delimiter. One of the followings: #\, #\. #\( #\) #\;
<string> Regular identifier
(delimited <string>) Delimited identifier
```
(parameter <num>)  Positional parameter (?)
(parameter <string>)  Named parameter (:foo)
(string  <string>)  Character string literal
(number  <string>)  Numeric literal
(bitstring <string>)  Binary string. <string> is like "01101"
(hexstring <string>)  Binary string. <string> is like "3AD20"

If it encounters an untokenizable string, it raises an `<sql-parse-error>` condition.

<sql-parse-error>  [Condition Type]
{text.sql} A condition to indicate an SQL parse error. Inherits `<error>`.

sql-string  [Instance Variable of `<sql-parse-error>`]
Holds the source SQL string.

12.78  text.template - Simple template expander

text.template  [Module]
This module lifts Gauche's built-in string interpolation feature to be more general template engine.

Gauche's string interpolation syntax is expanded at read time and then handled by macro expanders, and becomes a simple Scheme code fragment. For example, if you have this:

(let ([x 10])
  "The square of x is (* x x)."
)

It is eventually converted to this after macro expansion:

(let ([x 10])
  (string-append "The square of x is " (x->string (* x 0 x 0)) ".")
)

It is a kind of template expansion, but you have to have the template string as a literal, so it's restricted. With this module, you can feed template string and the bindings of the value at the runtime:

(define *template* "The square of x is (* x x).")

(expand-template-string *template* (make-template-environment :bindings '(x 10)))
⇒ "The square of x is 100."

The syntax of template strings is the same as string interpolation (see Section 6.11.4 [String interpolation], page 173); that is, tokens following `~` is read as a Scheme expression. In case if the token is a symbol and you need to delimit it from subsequent characters, you can use symbol escape by `|`.

You also need to provide a template environment, where the expressions in the template is evaluated. Note that, unlike string interpolation, those expressions can't refer to the local bindings.

expand-template-string template env  [Function]
Expands a template string template with a template environment env, and returns the result string.

expand-template-file filename env :optional paths  [Function]
Reads a template string from a file named by filename, expands it with a template environment env, and returns the result string.

If filename is not an absolute path, it is looked in the directories listed in paths.
**make-template-environment** :key extends imports bindings  

[Function]

Creates and returns a template environment. A template environment is like a module (see Section 4.13 [Modules], page 78): It maps symbols to values, and it can import bindings from other modules, or extend other modules.

The keyword arguments *extends* and *imports* must be a list of symbols; they specify names of modules to inherit from or to import from.

The keyword arguments *bindings* must either be a dictionary (anything that inherits `<dictionary>`), or a key-value list. The mappings represented by it are incorporated to the environment.

---

**12.79 text.tr - Transliterate characters**

**text.tr**  

[Module]

This module implements a transliterate function, that substitutes characters of the input string. This functionality is realized in Unix `tr(1)` command, and incorporated in various programs such as `sed(1)` and `perl`.

Gauche’s `tr` is aware of multibyte characters.

**tr** from-list to-list :key :complement :delete :squeeze :table-size :input  

:output

{text.tr} Reads from *input* and writes to *output*, with transliterating characters in *from-list* to the corresponding ones in *to-list*. Characters that doesn’t appear in *from-list* are passed through.

The default values of *input* and *output* are current input port and current output port, respectively.

Both *from-list* and *to-list* must be strings. They may contain the following special syntax. Other characters that doesn’t fits in the syntax are taken as they are.

- **x-y**  
  Expanded to the increasing sequence of characters from x to y, inclusive. The order is determined by the internal character encoding system; generally it is safer to limit use of this within the range of the same character class. The character x must be before y.

- **x*n**  
  Repeat x for n times. n is a decimal number notation. Meaningful only in *to-list*; it is an error to use this form in *from-list*. If n is omitted or zero, x is repeated until *to-list* matches the length of *from-list* (any character after it is ignored).

- **\x**  
  Represents x itself. Use this escape to avoid a special character to be interpreted as itself. Note that if you place a backslash in a string, you must write `\\`, for the Scheme reader also interprets backslash as a special character.

  There’s no special sequence to represent non-graphical characters, for you can put such characters by the string syntax.

Here’s some basic examples.

```scheme
;; swaps case of input
(tr "A-Za-z" "a-zA-Z")

;; replaces 7-bit non-graphical characters to ‘?’
(tr "\x00-\x19\x7f" "?\")
```

If *to-list* is shorter than *from-list*, the behavior depends on the keyword argument *delete*. If a true value is given, characters that appear in *from-list* but not in *to-list* are deleted. Otherwise, the extra characters in *from-list* are just passed through.
When a true value is specified to `complement`, the character set in `from-list` is complemented. Note that it implies huge set of characters, so it is not very useful unless either output character set is a single character (using `*`) or used with `delete` keyword.

When a true value is specified to `squeeze`, the sequence of the same replaced characters is squeezed to one. If `to-list` is empty, the sequence of the same characters in `from-list` is squeezed.

Internally, `tr` builds a table to map the characters for efficiency. Since Gauche can deal with potentially huge set of characters, it limits the use of the table for only smaller characters (<256 by default). If you want to transliterate multibyte characters on the large text, however, you might want to use larger table, trading off the memory usage. You can specify the internal table size by `table-size` keyword argument. For example, if you transliterate lots of EUC-JP hiragana text to katakana, you may want to set table size greater than 42483 (the character code of the last katakana).

Note that the pre-calculation to build the transliterate table needs some overhead. If you want to call `tr` many times inside loop, consider to use `build-transliterator` described below.

```scheme
(string-tr string from-list to-list :key :complement :delete :squeeze :table-size)

{text.tr} Works like `tr`, except that input is taken from a string `string`.

(build-transliterator from-list to-list :key :complement :delete :squeeze :table-size :input :output)

{text.tr} Returns a procedure that does the actual transliteration. This effectively “pre-compiles” the internal data structure. If you want to run `tr` with the same sets repeatedly, you may build the procedure once and apply it repeatedly, saving the overhead of initialization.

A note for an edge case: When `input` and/or `output` keyword arguments are omitted, the created transliterator is set up to use current-input-port and/or current-output-port at the time transliterator is called.

```scheme
(with-input-from-file "huge-file.txt"
  (lambda ()
    (let loop ((line (read-line)))
      (unless (eof-object? line) (tr "A-Za-z" "a-zA-Z"))))))

;; runs more efficiently...

(with-input-from-file "huge-file.txt"
  (lambda ()
    (let ((ptr (build-transliterator "A-Za-z" "a-zA-Z")))
      (let loop ((line (read-line)))
        (unless (eof-object? line) (ptr))))))
```

### 12.80 text.tree - Lazy text construction

**text.tree**

Defines simple but commonly used functions for a text construction.

When you generate a text by a program, it is a very common operation to concatenate text segments. However, using `string-append` repeatedly causes unnecessary copying of intermediate strings, and sometimes such intermediate strings are discarded due to the error situation (for example, think about constructing an HTML document in the CGI script).
The efficient technique is to delay concatenation of those text segments until it is needed. In Scheme it is done very easily by just consing the text segments together, thus forming a tree of text, and then traverse the tree to construct a text. You can even directly write out the text during traversal, avoiding intermediate string buffer. (Hans Boehm’s “cord” library, which comes with his garbage collector library, uses this technique and proves it is very efficient for editor-type application).

Although the traversal of the tree can be written in a few lines of Scheme, I provide this module in the spirits of OnceAndOnlyOnce. Also it’s easier if we have a common interface.

**write-tree**  
*tree :optional out*  
{text.tree} Writes out an *tree* as a tree of text, to the output port *out*. If *out* is omitted, the current output port is used.

Two methods are defined for this generic function, as shown below. If you have more complex behavior, you can define more methods to customize the behavior.

**write-tree** ((tree <list>) *out*)  
**Method**

**write-tree** ((tree <top>) *out*)  
**Method**

{text.tree} Default methods. For a list, *write-tree* is recursively called for each element. Any objects other than list is written out using display.

**tree->string**  
*tree*  
{text.tree} Just calls the *write-tree* method for *tree* using an output string port, and returns the result string.

### 12.81 util.combinations - Combination library

util.combinations  
[Module]

This module implements several useful procedures of combinations, permutations and related operations.

Most procedures in the module have two variants: a procedure without star (e.g. *permutations*) treats all elements in the given set distinct, while a procedure with star (e.g. *permutations*-) considers duplication. The procedures with star take optional *eq* argument that is used to test equality, which defaults to *eqv?*.

**permutations**  
*set*  
[Function]

**permutations**  
*set :optional eq*  
[Function]

{util.combinations} Returns a list of all permutations of a list *set*.

\[
\text{(permutations '}(a b c)) \\
\Rightarrow ((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))
\]

\[
\text{(permutations '}(a a b)) \\
\Rightarrow ((a a b) (a b a) (a a b) (a b a) (b a a) (b a a))
\]

\[
\text{(permutations-'} (a a b) \\
\Rightarrow ((a a b) (a b a) (b a a))
\]

The number of possible permutations explodes if *set* has more than several elements. Use with care. If you want to process each permutation at a time, consider *permutations-for-each* below.

**permutations-for-each**  
*proc set*  
[Function]

**permutations-’for-each**  
*proc set :optional eq*  
[Function]

{util.combinations} For each permutation of a list *set*, calls *proc*. Returns an undefined value.
Chapter 12: Library modules - Utilities

**combinations** set n

**combinations** set n :optional eq
t {util.combinations} Returns a list of all possible combinations of n elements out of a list set.

(combinations '(a b c) 2)
⇒ ((a b) (a c) (b c))

(combinations* '(a a b) 2)
⇒ ((a a) (a b) (a b))

Watch out the explosion of combinations when set is large.

**combinations-for-each** proc set n

**combinations**-for-each proc set n :optional eq
t {util.combinations} Calls proc for each combination of n elements out of set. Returns an undefined value.

**power-set** set

**power-set** set :optional eq
t {util.combinations} Returns power set (all subsets) of a list set.

(power-set '(a b c))
⇒ (( ) (a) (b) (c) (a b) (a c) (b c) (a b c))

(power-set* '(a a b))
⇒ (( ) (a) (a a) (a b) (a a b))

**power-set-for-each** proc set

**power-set**-for-each proc set :optional eq
t {util.combinations} Calls proc for each subset of set.

**power-set-binary** set

{util.combinations} Returns power set of set, like power-set, but in different order. Power-set-binary traverses subset space in depth-first order, while power-set in breadth-first order.

(power-set-binary '(a b c))
⇒ (( ) (c) (b c) (a) (a c) (a b) (a b c))

**cartesian-product** list-of-sets

**cartesian-product**-right list-of-sets
t {util.combinations} Returns a cartesian product of sets in list-of-sets. Cartesian-product construct the result in left fixed order (the rightmost element varies first), while cartesian-product-right in right fixed order (the leftmost element varies first).

(cartesian-product '((a b c) (0 1)))
⇒ ((a 0) (a 1) (b 0) (b 1) (c 0) (c 1))

(cartesian-product-right '((a b c) (0 1)))
⇒ ((a 0) (b 0) (c 0) (a 1) (b 1) (c 1))
12.82 util.digest - Message digester framework

util.digest

This module provides a base class and common interface for message digest algorithms, such as MD5 (see Section 12.50 [MD5 message digest], page 939) and SHA (see Section 12.53 [SHA message digest], page 946).

A typical way to compute a message digest would be as follows:

```scheme
(use rfc.sha) ; import actual digest implementation
(use util.digest)

(digest-message-to 'base64url <sha256> "message")
```

You can use different algorithms, and choose different output format (e.g. u8vector, hexadecimal string, etc.) If you need to digest with a secret key, see Section 12.45 [HMAC keyed-hashing], page 928.

User API

To digest a message, you can use the following API.

```
digest-message-to target digester message
```

This function computes a digest of `message` using algorithm implemented by `digester`, and returns as an instance of class `target`.

The `message` argument may be a string or a u8vector.

The `digester` argument must be a class inheriting `<message-digest-algorithm>` and implementing the digester protocol described below. Gauche provides SHA (see Section 12.53 [SHA message digest], page 946) and MD5 (see Section 12.50 [MD5 message digest], page 939) by default.

The supported `target` class is `<u8vector>` and `<string>`. Digest result is a bytestring in general, so using `<u8vector>` is recommended; if you give `<string>`, the result is likely to be an incomplete string.

For the convenience, the following symbols are also allowed as `target`, in which case the output byte sequence is encoded to a readable string. For the details of those encodings, see Section 12.42 [Base64 encoding/decoding], page 921.

- `base64` Base64 encoding (original).
- `base64url` URL-safe Base64 encoding.
- `base64url-nopad` URL-safe Base64 encoding and omitting trailing padding character (=)
- `base32` Base32 encoding.
- `base32hex` Base32hex encoding.
- `base16` Base16 encoding, a.k.a. hexadecimal encoding (using #[0-9A-F]).
- `hex` Base16 encoding but use lowercase letters (using #[0-9a-f]).

```
digest-to target digester
```

This function reads input from the current input port until EOF, compute its digest using the algorithm specified by `digester`, and returns result in an instance of a class given to `target`. You can also give some symbols to specify encodings as `target`. See `digest-message-to` above for the details of `target` and `digester`. {util.digest}
Implementer API

The following API is for the implementer of the actual digest algorithms.

<message-digest-algorithm-meta> [Class]
{util.digest} A metaclass of message digest algorithm implementation.

hmac-block-size [Instance Variable of <message-digest-algorithm-meta>]
Specifies the block size (in bytes), which is specific to each algorithm. (This is a slot for each class object that implements the algorithm, not for instance of such classes. Only the author of such digest classes needs to care. See ext/digest/sha.scm in the source tree for more details.)

<message-digest-algorithm> [Class]
{util.digest} A base class of message digest algorithm implementation.

The concrete subclass of message digest algorithm has to implement the following methods.

digest-update! algorithm data [Generic function]
{util.digest} Takes the instance of message-digest algorithm, and updates it with the data data, which can be either a u8vector or a (possibly incomplete) string.

digest-final! algorithm [Generic function]
{util.digest} Finalizes the instance of message-digest algorithm, and returns the digest result in an incomplete string.

digest class [Generic function]
{util.digest} A wrapper of digest routines. Given message-digest algorithm class, this function reads the input data from current input port until EOF, and returns the digest result in an incomplete string.

Deprecated API

digest-string class string [Generic function]
{util.digest} Deprecated. Use digest-message-to for the new code.
Same as (digest-message-to <string> class string), where only a string is allowed to the string argument.

digest-hexify digest-result [Function]
{util.digest} Deprecated. Same as (base16-encode-message digest-result :lowercase #t) (see Section 12.42 [Base64 encoding/decoding], page 921). In typical case, you can pass hex as the target to digest-message-to to get the hexified output directly. This procedure is only kept for the backward compatibility.

12.83 util.dominator - Calculate dominator tree

util.dominator [Module]
Dominator tree is an auxiliary structure for control flow graphs. It is frequently used in the flow analysis of compilers, but also useful for handling general directed graphs.

calculate-dominators start upstreams downstreams node-comparator [Function]
{util.dominator} The four arguments represent a directed, possibly cyclic, graph. Here, we use Node to denote an abstract type of a node of the graph. It can be anything—the algorithm is oblivious on the actual type of nodes.

start :: Node
The start node, or the enter node, of the graph.
upstreams :: Node -> (Node ...)
A procedure that takes a node, and returns its upstream (immediate ancestor) nodes.

downstreams :: Node -> (Node ...)
A procedure that takes a node, and returns its downstream (immediate descendant) nodes.

node-comparator
A comparator that is used to determine if two nodes are equal to each other. It doesn’t need to have comparison procedure (we don’t need to see which is smaller than the other), but it has to have hash function, for we use hashtables internally. (See Section 6.2.4 [Basic comparators], page 116, for the details of comparators.)

The procedure returns a list of (node1 node2), where node2 is the immediate dominator of node1.

If there are node in the given graph that are unreachable from start, such nodes are ignored and not included in the result.

(A bit of explanation: Suppose you want to go to node X from start. There may be multiple routes, but if you have to pass node Y no matter which route you take, then Y is a dominator of X. There may be many dominators of X. Among them, there’s always one dominator such that all other X’s dominators are also its dominators—in other words, the closest dominator of X—which is called the immediate dominator of X.)

Let’s see an example. You have this directed graph:

```
A (start)
|   
| V
B <---------+
|   |   
|   |   
|   v v
C -------> D ---+
|   |   
|   v v
E <------- F
```

Let’s represent the graph by a list of (x y z ...) where x can directly go to either y z ... .

```
(define *graph* '((A B)  
                   (B C D)  
                   (C D E)  
                   (D F B)  
                   (F E)))
```

Then you can calculate the immediate dominator of each node as follows:

```
(calculate-dominators 'A  
            (^n (filter-map (^g (and (memq n (cdr g)) (car g))) *graph*))  
            (^n (assoc-ref *graph* n '()))  
            eq-comparator)  
      ⇒ ((E B) (F D) (D B) (C B) (B A))
```

That is, E’s immediate dominator is B, F’s is D, and so on.

The result itself can be viewed as a tree. It is called a dominator tree.

F
12.84 util.isomorph - Determine isomorphism

util.isomorph

Provides a procedure that determines whether two structures are isomorphic.

isomorphic? obj1 obj2 :optional context

{util.isomorph} Returns #t if obj1 and obj2 are isomorphic.
context is used if you want to call isomorphic? recursively inside object-isomorphic? described below.

(isomorphic? '(a b) '(a b)) ⇒ #t
(define x (cons 0 0))
(define y (cons 0 0))
(isomorphic? (cons x x) (cons x y)) ⇒ #f
(isomorphic? (cons x x) (cons y y)) ⇒ #t

object-isomorphic? obj1 obj2 context

{util.isomorph} With this method, you can customize how to determine isomorphism of two objects. Basically, you will call isomorphic? recursively for each slots of object you want to traverse; the method should return #t if all of the test succeeds, or return #f otherwise. context is an opaque structure that keeps the traversal context, and you should pass it to isomorphic? as is.
The default method returns #t if obj1 and obj2 are equal (in the sense of equal?).

12.85 util.lcs - The longest common subsequence

util.lcs

This module implements the algorithm to find the longest common subsequence of two given sequences. The implemented algorithm is based on Eugene Myers’ O(ND) algorithm (Eugene Myers, An O(ND) Difference Algorithm and Its Variations, Algorithmica Vol. 1 No. 2, pp. 251-266, 1986.).

One of the applications of this algorithm is to calculate the difference of two text streams; see Section 12.65 [Calculate difference of text streams], page 991.

lcs seq-a seq-b :optional eq-fn

{util.lcs} Calculates and returns the longest common sequence of two lists, seq-a and seq-b. Optional eq-fn specifies the comparison predicate; if omitted, equal? is used.
(lcs '(x a b y) '(p a q b)) ⇒ (a b)
Chapter 12: Library modules - Utilities

\textbf{lcs-with-positions} \textit{seq-a seq-b} :optional eq-fn \hspace{1cm} \textbf{[Function]}
\{util.lcs\} This is the detailed version of \textit{lcs}. The arguments are the same.

Returns a list of the following structure:
\[
(\text{length} \ ((\text{elt} \ a\text{-pos} \ b\text{-pos}) \ . . .))
\]

\textit{Length} is an integer showing the length of the found LCS. What follows is a list of elements of LCS; each sublist consists of the element, the integer position of the element in \textit{seq-a}, then the integer position of the element in \textit{seq-b}.

- \(\text{lcs-with-positions} \ (\text{'(a)} \ \text{'(a)})\)
  \Rightarrow (1 \ ((\text{a} 0 0)))

- \(\text{lcs-with-positions} \ (\text{'(x a b y)} \ \text{'(p q a b)})\)
  \Rightarrow (2 \ ((\text{a} 1 2) \ (\text{b} 2 3)))

- \(\text{lcs-with-positions} \ (\text{'(x a b y)} \ \text{'(p a q b)})\)
  \Rightarrow (2 \ ((\text{a} 1 1) \ (\text{b} 2 3)))

- \(\text{lcs-with-positions} \ (\text{'(x y)} \ \text{'(p q)})\)
  \Rightarrow (0 \ ())

\textbf{lcs-fold} \textit{a-proc b-proc both-proc seed a b} :optional eq-fn \hspace{1cm} \textbf{[Function]}
\{util.lcs\} A fundamental iterator over the "edit list" derived from two lists \textit{a} and \textit{b}.

\textit{A-proc}, \textit{b-proc}, \textit{both-proc} are all procedures that take two arguments. The second argument is an intermediate state value of the calculation. The first value is an element only in \textit{a} for \textit{a-proc}, or an element only in \textit{b} for \textit{b-proc}, or an element in both \textit{a} and \textit{b} for \textit{both-proc}.

The return value of each procedure is used as the state value of the next call of either one of the procedures. \textit{Seed} is used as the initial value of the state value. The last state value is returned from \textit{lcs-fold}.

The three procedures are called in the following order: Suppose the sequence \textit{a} consists of \textit{a'ca''}, and \textit{b} consists of \textit{b'cb''}, where \textit{a'}, \textit{b'}, \textit{a''}, and \textit{b''} are subsequences, and \textit{c} is the head of the LCS of \textit{a} and \textit{b}. Then \textit{a-proc} is called first on each element in \textit{a'}, \textit{b-proc} is called second on each element in \textit{b'}, then \textit{both-proc} is called on \textit{c}. Afterwards, the process is repeated using \textit{a''} and \textit{b''}.

\textbf{lcs-edit-list} \textit{a b} :optional eq-fn \hspace{1cm} \textbf{[Function]}
\{util.lcs\} Calculates 'edit-list' from two lists \textit{a} and \textit{b}, which is the smallest set of commands (additions and deletions) that changes \textit{a} into \textit{b}. This procedure is built on top of \textit{lcs-fold} above.

Returns a list of \textit{hunks}, which is a contiguous section of additions and deletions. Each hunk consists of a list of directives, which is a form of:

\[
(+|\text{-} \ \text{position} \ \text{element})
\]

Here's an example. Suppose \textit{a} and \textit{b} are the following lists, respectively.

\[
\text{a} \equiv \ ("A" \ "B" \ "C" \ "E" \ "H" \ "J" \ "L" \ "M" \ "N" \ "P")
\]

\[
\text{b} \equiv \ ("B" \ "C" \ "D" \ "E" \ "F" \ "J" \ "K" \ "L" \ "M" \ "R" \ "S" \ "T")
\]

Then, \(\text{lcs-edit-list} \ \text{a b equal?}\) returns the following list.

\[
(((\text{-} \ 0 \ "A")\))
(+(\text{+} \ 2 \ "D")\))
(\text{-} \ 4 \ "H") \ (+(\text{+} \ 4 \ "F")\))
(+(\text{+} \ 6 \ "K")\))
(\text{-} \ 8 \ "N") \ (\text{-} \ 9 \ "P") \ (+(\text{+} \ 9 \ "R") \ (+(\text{+} \ 10 \ "S") \ (+(\text{+} \ 11 \ "T")\}))
\]
The result consists of five hunks. The first hunk consists of one directive, (− 0 "A"), which means the element "A" at the position 0 of list a has to be deleted. The second hunk also consists of one directive, (+ 2 "D"), meaning the element "D" at the position 2 of list b has to be added. The third hunk means "H" at the position 4 of list a should be removed and "F" at the position 4 of list b should be added, and so on.

If you are familiar with Perl's Algorithm::Diff module, you may notice that this is the same structure that its diff procedure returns.

\[\text{lcs-edit-list/context} \quad a \ b : \text{optional eq-fn :key context-size} \quad \text{[Function]}\]
\[\text{lcs-edit-list/unified} \quad a \ b : \text{optional eq-fn :key context-size} \quad \text{[Function]}\]

{util.lcs} Calculates edit list of two sequences \(a\) and \(b\), including context surrounding the changes. It is similar to what you get with \text{diff -c} and \text{diff -u} commands, respectively. The text.diff module provides context diff on top of this procedure (see Section 12.65 [Calculate difference of text streams], page 991).

The \text{eq-fn} argument specifies the equality predicate on the elements. Its default is \text{equal?}. The \text{context-size} keyword argument specifies the maximum size of the unchanged elements surrounding each edits. It must be an exact positive integer. Its default is 3.

Returns a list of hunks. Each hunk represents a chunk of difference between \(a\) and \(b\). The format of hunk differs between \text{lcs-edit-list/context} and \text{lcs-edit-list/unified}.

A hunk of \text{lcs-edit-list/context} has the following form:

\[
\begin{align*}
&\langle \text{hunk} \rangle : \#(\langle \text{a-edits} \rangle \ \langle \text{b-edits} \rangle) \\
&\langle \text{a-edits} \rangle : \langle \text{edits} \rangle \\
&\langle \text{b-edits} \rangle : \langle \text{edits} \rangle \\
&\langle \text{edits} \rangle : (\langle \text{start-pos} \rangle \ \langle \text{end-pos} \rangle \ \langle \text{edit} \rangle \ \ldots) \\
&\langle \text{edit} \rangle : (= \langle \text{element} \rangle) \quad ; \text{same in both } a \text{ and } b \\
&| (- \langle \text{element} \rangle) \quad ; \text{deleted: only in } a \\
&| (+ \langle \text{element} \rangle) \quad ; \text{inserted: only in } b \\
&| (! \langle \text{element} \rangle) \quad ; \text{changed}
\end{align*}
\]

The \text{start-pos} and \text{end-pos} are exact integer indexes of the sequence the hunk is covering; the start position inclusive, the end position exclusive, and 0-based.

Each \text{edit} shows the element in that range, and whether the element is to be unchanged, deleted, inserted or changed, to edit the sequence \(a\) to make \(b\).

Here's an example:

\[
\begin{align*}
&\text{lcs-edit-list/context } '(a \ b \ c) \ '(a \ B \ c \ d) \\
&\Rightarrow \\
&(\#((0 \ 3 \ (= \ a) \ (! \ b) \ (= \ c)) \\
&(0 \ 4 \ (= \ a) \ (! \ B) \ (= \ c) \ (+ \ d))))
\end{align*}
\]

The result contains one hunk. It shows that you should change \(b\) to \(B\), and insert \(d\).

A hunk of \text{lcs-edit-list/unified} has the following form:

\[
\begin{align*}
&\langle \text{hunk} \rangle : \#(\langle \text{a-start} \rangle \ \langle \text{a-size} \rangle \ \langle \text{b-start} \rangle \ \langle \text{b-size} \rangle \ \langle \text{edit} \rangle \ \ldots) \\
&\langle \text{edit} \rangle : (= \langle \text{element} \rangle) \quad ; \text{unchanged} \\
&| (- \langle \text{element} \rangle) \quad ; \text{deleted} \\
&| (+ \langle \text{element} \rangle) \quad ; \text{inserted}
\end{align*}
\]

Each \text{start} and \text{size} in the hunk specifies the start position within the sequence (0-based) and the length of the hunk in the sequence, respectively.

\[
\begin{align*}
&(\text{lcs-edit-list/unified } '(a \ b \ c) \ '(a \ B \ c \ d))
\end{align*}
\]
⇒
(#(0 3 0 4 ((= a) (- b) (+ B) (= c) (+ d))))

**Context size:** In each hunk, up to context-size unchanged elements are attached before and after the inserted/delete/changed elements, to show the context. If there’s less than (+ 1 (\(*\) context-size 2)) elements between two changes, they are merged into one hunk. In the followig example, the change of b to B and e to E are merged into one hunk for they’re not far enough, while the change of i to I is in a separate hunk:

(lcs-edit-list/context '(a b c d e f g h i j)
  '(a B c d E f g h I j)
  equal? :context-size 1)
⇒
(#((0 6 (= a) (! b) (= c) (= d) (! e) (= f)))
 (0 6 (= a) (! B) (= c) (= d) (! E) (= f)))
 #((7 10 (= h) (! i) (= j))
 (7 10 (= h) (! I) (= j))))

(lcs-edit-list/unified '(a b c d e f g h i j)
  '(a B c d E f g h I j)
  equal? :context-size 1)
⇒
(#(0 0 6 6 ((= a) (- b) (+ B) (= c) (= d) (- e) (+ E) (= f)))
 #(7 3 7 3 ((= h) (- i) (+ I) (= j))))

**12.86 util.levenshtein - Levenshtein edit distance**

**util.levenshtein** [Module]

This module provides procedures to calculate edit distance between two sequences. Edit distance is the minimum number of edit operations required to match one sequence to another. Three algorithms are implemented:

Levenshtein distance

Count deletion of one element, insertion of one element, and substitution of one element.

Damerau-Levenshtein distance

Besides deletion, insertion and substitution, we allow transposition of adjacent elements.

Restricted edit distance

Also called optimal string alignment distance. Like Damerau-Levenshtein, but once transposition is applied, no further editing on those elements are allowed.

These algorithms are often used to compare strings, but the procedures in this module can handle any type of sequence (see Section 9.31 [Sequence framework], page 510).

l-distance seq-A seq-B :key elt= cutoff

l-distances seq-A seq-Bs :key elt= cutoff

re-distance seq-A seq-B :key elt= cutoff

re-distances seq-A seq-Bs :key elt= cutoff

dl-distance seq-A seq-B :key elt= cutoff

dl-distances seq-A seq-Bs :key elt= cutoff

{util.levenshtein} Calculates Levenshtein distance (l-\*), restricted edit distance (re-\*) and Damerau-Levenshtein distance (dl-\*) between sequences, respectively. Each algorithm comes in two flavors: The singular form \*distance takes two sequences, seq-A and seq-B,
and calculates distance between them. The plural form `-distances` takes a sequence `seq-A` and a list of sequences `seq-Bs`, and calculates distances between `seq-A` and each in `seq-Bs`. If you need to calculate distances from a single sequence to many sequences, using the plural version is much faster than repeatedly calling the singular version, for the plural version can reuse internal data structures and save allocation and setup time.

Sequences can be any object that satisfy the `<sequence>` protocol (see Section 9.31 [Sequence framework], page 510).

The keyword argument `elt=` is used to compare elements in the sequences. Its default is `eqv?`.

The keyword argument `cutoff` must be, if given, a nonnegative exact integer. Once the possible minimum distance between two sequences becomes greater than this number, the algorithm stops and gives `#f` as the result, and moves on to the next calculation. This is useful when you run the algorithm on large set of sequences and you only need to look for the pairs closer than the certain limit.

In our implementation, Levenshtein is the fastest, Damerau-Levenshtein is the slowest and Restricted edit is somewhere inbetween. If you don’t need to take into account of transpositions, use Levenshtein: it counts 2 for `cat` -> `act`, while other algorithms yield 1 for it. If you need to consider transpositions, choose either `re-` or `dl-`. The catch in `re-` is that it does not satisfy triangular inequality, i.e. for given three sequences X, Y and Z, (Damerau-)Levenshtein distance L always satisfy \( L(X;Z) \leq L(X;Y) + L(Y;Z) \), but restricted edit distance doesn’t guarantee that.

\[
\text{(l-distance "cat" "act") } \Rightarrow 2 \\
\text{(l-distances "cat" '("Cathy" "scathe" "stack")} \\
: \text{elt= char-ci=?)} \\
\Rightarrow (2 3 4)
\]

\[
\text{(re-distance "cat" "act") } \Rightarrow 1
\]

\[
\text{(re-distances "pepper"} \\
'("peter" "piper" "picked" "peck" "pickled" "peppers")} \\
: \text{cutoff 4}) \\
\Rightarrow (2 2 4 4 #f 1)
\]

\[
\text{(dl-distance '(a b c d e) '(c d a b e)) } \Rightarrow 4
\]

Note: If you pass a list of sequences to the second argument of the singular version by accident, you might not get an error immediately because a list is also a sequence.

12.87 util.match - Pattern matching

util.match

This module is a port of Andrew Wright’s pattern matching macro library. It is widely used in the Scheme world, and ported to various Scheme implementations, including Chez Scheme, PLT Scheme, Scheme48, Chicken, and SLIB. It is similar to, but more powerful than Common Lisp’s `destructuring-bind`.

This version retains compatibility of the original Wright’s macro, except (1) `box` is not supported since Gauche doesn’t have one, and (2) structure matching is integrated to Gauche’s object system.

We show a list of APIs first, then the table of complete syntax of patterns, followed by examples.
Pattern matching API

**match expr clause ...**

{util.match} Each clause is either one of the followings:

- `(pat body ...)`
- `(pat (⇒ identifier) body ...)`

First, the `expr` is matched against `pat` of each clauses. The detailed syntax of the pattern is explained below.

If a matching `pat` is found, the *pattern variables* in `pat` are bound to the corresponding elements in `expr`, then `body ...` are evaluated. Then `match` returns the value(s) of the last expression of `body ...`.

If the clause is the second form, `identifier` is also bound to the failure continuation of the clause. It is a procedure with no arguments, and when called, it jumps back to the matcher as if the matching of `pat` is failed, and `match` continues to try the rest of clauses. So you can perform extra tests within `body ...` and if you’re not satisfied you can reject the match by calling `(identifier)`. See the examples below for more details.

If no `pat` matches, `match` reports an error.

**match-lambda clause ...**

{util.match} Creates a function that takes one argument and performs `match` on it, using `clause ...`. It’s functionally equivalent to the following expression:

- `(lambda (expr) (match expr clause ...))`

Example:

```scheme
(map (match-lambda
       ((item price-per-lb (quantity ’lbs))
        (cons item (* price-per-lb quantity)))
       ((item price-per-lb (quantity ’kg))
        (cons item (* price-per-lb quantity 2.204)))
       ’((apple 1.23 (1.1 lbs))
       (orange 0.68 (1.4 lbs))
       (cantaloupe 0.53 (2.1 kg))))
⇒ ’((apple . 1.353) (orange . 0.952)
   (cantaloupe . 2.4530520000000005))
```

**match-lambda* clause ...**

{util.match} Like `match-lambda`, but performs `match` on the list of whole arguments. It’s functionally equivalent to the following expression:

- `(lambda expr (match expr clause ...))`

**match-let ((pat expr) ...) body-expr ...**

{util.match} Generalize `let`, `let*`, and `letrec` to allow patterns in the binding position rather than just variables. Each `expr` is evaluated, and then matched to `pat`, and the bound pattern variables are visible in `body-expr` ...

- `(match-let ( (((ca . cd) ...) ’((a . 0) (b . 1) (c . 2)))
       (list ca cd))
  ⇒ ((a b c) (0 1 2)))
```

If you’re sick of parenthesis, try `match-let1` below.
match-let1 pat expr body-expr ...  
{util.match} This is a Gauche extension and isn’t found in the original Wright’s code. This one is equivalent to the following code:

\[
\text{(match-let ((pat expr)) body-expr ...)}
\]

Syntactically, match-let1 is very close to the Common Lisp’s destructuring-bind.

\[
\text{(match-let1 ('let ((var val) ...) body ...)}
\]

\[
\text{'(let ((a b) (c d)) foo bar baz)}
\]

⇒ \((a c) (b d) (foo bar baz))

match-define pat expr

{util.match} Like toplevel define, but allows a pattern instead of variables.

\[
\text{(match-define (x . xs) (list 1 2 3))}
\]

\[
x ⇒ 1 \\
xs ⇒ (2 3)
\]

Pattern syntax

Here’s a summary of pattern syntax. The asterisk (\(*\)) after explanation means Gauche’s extension which does not present in the original Wright’s code.

<table>
<thead>
<tr>
<th>pat</th>
<th>patvar</th>
<th>;;; anything, and binds pattern var</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_</td>
<td>;;; anything</td>
</tr>
<tr>
<td></td>
<td>()</td>
<td>;;; the empty list</td>
</tr>
<tr>
<td></td>
<td>#t</td>
<td>;;; #t</td>
</tr>
<tr>
<td></td>
<td>#f</td>
<td>;;; #f</td>
</tr>
<tr>
<td></td>
<td>string</td>
<td>;;; a string</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td>;;; a number</td>
</tr>
<tr>
<td></td>
<td>character</td>
<td>;;; a character</td>
</tr>
<tr>
<td></td>
<td>'sexp</td>
<td>;;; an s-expression</td>
</tr>
<tr>
<td></td>
<td>'symbol</td>
<td>;;; a symbol (special case of s-expr)</td>
</tr>
<tr>
<td>(pat1 ... patN)</td>
<td>;;; list of n elements</td>
<td></td>
</tr>
<tr>
<td>(pat1 ... patN . patN+1)</td>
<td>;;; list of n or more</td>
<td></td>
</tr>
<tr>
<td>(pat1 ... patN patN+1 ooo)</td>
<td>;;; list of n or more, each element</td>
<td></td>
</tr>
<tr>
<td></td>
<td>;;; of remainder must match patN+1</td>
<td></td>
</tr>
<tr>
<td>#(pat1 ... patN)</td>
<td>;;; vector of n elements</td>
<td></td>
</tr>
<tr>
<td>#(pat1 ... patN patN+1 ooo)</td>
<td>;;; vector of n or more, each element</td>
<td></td>
</tr>
<tr>
<td></td>
<td>;;; of remainder must match patN+1</td>
<td></td>
</tr>
<tr>
<td>($ class pat1 ... patN)</td>
<td>;;; an object (patK matches in slot order)</td>
<td></td>
</tr>
<tr>
<td>(struct class pat1 ... patN)</td>
<td>;;; ditto (*)</td>
<td></td>
</tr>
<tr>
<td>(@ class (slot1 pat1) ...)</td>
<td>;;; an object (using slot names) (*)</td>
<td></td>
</tr>
<tr>
<td>(object class (slot1 pat1) ...)</td>
<td>;;; ditto (*)</td>
<td></td>
</tr>
<tr>
<td>(= proc pat)</td>
<td>;;; apply proc, match the result to pat</td>
<td></td>
</tr>
<tr>
<td>(and pat ...)</td>
<td>;;; if all of pats match</td>
<td></td>
</tr>
<tr>
<td>(or pat ...)</td>
<td>;;; if any of pats match</td>
<td></td>
</tr>
<tr>
<td>(not pat ...)</td>
<td>;;; if all pats don’t match at all</td>
<td></td>
</tr>
<tr>
<td>(? predicate pat ...)</td>
<td>;;; if predicate true and all pats match</td>
<td></td>
</tr>
<tr>
<td>(set! patvar)</td>
<td>;;; anything, and binds setter</td>
<td></td>
</tr>
<tr>
<td>(get! patvar)</td>
<td>;;; anything, and binds getter</td>
<td></td>
</tr>
<tr>
<td>'qp</td>
<td>;;; a quasi-pattern</td>
<td></td>
</tr>
</tbody>
</table>

patvar : a symbol except _, quote, $, struct, @, object, =, and, or,
Chapter 12: Library modules - Utilities

not, ?, set!, get!, quasiquote, ..., ___, ..k, __k.

- A bare symbol is a "pattern variable"; it matches anything, and the matched part of the expression is bound to the symbol. The following symbols have special meanings and cannot be used as a pattern variable: _, quote, $, struct, @, object, =, and, or, not, ?, set!, get!, quasiquote, ..., ___ and ..k and __k where k is an integer.
- A symbol _ matches anything, without binding a pattern variable. It can be used to show "don’t care" placeholder.
- Literals such as emptylist, booleans, strings, numbers, characters and keywords match the same object (in the sense of equal?).
- Quoted expression matches the same expression (in the sense of equal?). You can use a quoted symbol to match the symbol itself.
- A keyword, without a quote, used to match the same keyword object. Since we’re in the process of unifying keywords and symbols, the user is recommended to write keywords with a quote in a pattern in order to match the keyword in the input. See Section 6.8.1 [Keyword and symbol integration], page 158, for the details.
- A list and a vector in general match a list or a vector whose elements matches the elements in the pattern recursively, unless the first element of the list is one of the special symbols listed above, it has a special meaning.
- As a special case, the last element of a vector or a list can be followed by a symbol ___. In that case, the pattern just before the symbol ___ can be applied repeatedly until it consumes all the elements in the given expression. A symbol ___ can be used in place of ___; it is useful when you want to produce a pattern by syntax-rules macro.
- For a list pattern, you can also use a symbol ..1, ..2, ..., which specifies the minimum number of repetition.
- ($ class pat1 ...) matches an instance of a class class. Each pattern pat1 ... matches each value of slots, in order of (class-slots class) by default. (Records are exception; they match the same order as their default constructor since 0.9.6.)
- (struct class pat1 ...) has the same meaning. Although the original Wright’s code doesn’t have struct, PLT Scheme has it in its extended match feature, and it is more descriptive.

This is an adaptation of the original feature that can match structures. It is useful to match a simple instance that you know the order of slots; for example, a simple record created by define-record-type (see Section 9.28 [Record types], page 502) would be easy to match by positioned values.
- If the instance’s class uses inheritances, it is a bit difficult to match by positions. You can use @ or object pattern below to match using slot names.
- (object class (slot1 pat1) ...) matches an instance of a class class whose value of slot1 ... matches pat1 ... This is Gauche’s extension. @ can be used in place of object, but object is recommended because of descriptiveness.
• \((\mathbf{* proc pat})\) first applies \textit{proc} to the corresponding expression, then match the result with \textit{pat}.

• \((\mathbf{and pat \ldots})\) matches when all \textit{pat} matches the corresponding subtree. A common idiom to realize “as” matching, with which you can bind the entire tree as well as its substrees, can be written with \textit{and} operator. For example, \((\mathbf{and (x . y) p})\) matches a pair, binding its \texttt{cdr} to \texttt{x}, its \texttt{car} to \texttt{y}, and the entire pair to \texttt{p}.

• \((\mathbf{or pat \ldots})\) tries to match each \textit{pat} against the corresponding subtree, and if any succeeds, the entire \textit{or} pattern succeeds. If no \textit{pat} matches the subtree, the entire \textit{or} pattern fails. If there’re pattern variables in \textit{pat}, the set of variables in each \textit{pat} must be the same: That is, \((\mathbf{or (_x y) (_x y _)})\) is ok but \((\mathbf{or (_x x) (_x y _)})\) is not. It is because the latter may leave some of pattern variables unbound even match succeeds.

• \((\mathbf{not pat \ldots})\) matches when none of \textit{pat} matches the corresponding subtree.

• \((\mathbf{(? predicate pat \ldots)})\) first applies a predicate to the corresponding expression, and if it returns true, applies each \textit{pat} \ldots to the expression.

• \((\mathbf{set! patvar})\) matches anything, and binds an one-argument procedure to a pattern variable \textit{patvar}. If the procedure is called, it replaces the value of matched pattern for the given argument.

• \((\mathbf{get! patvar})\) matches anything, and binds a zero-argument procedure to a pattern variable \textit{patvar}. If the procedure is called, it returns the matched value.

• ‘\textit{qp}’ is a quasipattern. ‘\textit{qp}’ is quoted, in the sense that it matches itself, \textit{except} the pattern that is unquoted. (Don’t confuse quasipattern to quasiquote, though the functions are similar. Quasiquote turns off evaluation except unquoted subtree. Quasiquote turns off the special pattern syntax except unquoted subtree. See the examples below).

**Pattern examples**

A simple structure decomposition:

\[
\begin{align*}
\text{(match '}(0 (1 2) (3 4 5)) \\
\quad [(\text{a (b c) (d e f)}) \\
\quad (\text{list a b c d e f})]
\Rightarrow (0 1 2 3 4 5)
\end{align*}
\]

Using predicate patterns:

\[
\begin{align*}
\text{(match 123)} \\
\quad [(\text{(? string? x) (list 'string x)}) \\
\quad (\text{(? number? x) (list 'number x)})]
\Rightarrow (\text{number 123})
\end{align*}
\]

Extracting variables and expressions from \texttt{let}. Uses repetition and predicate patterns:

\[
\begin{align*}
\text{(define let-analyzer} \\
\quad (\text{match-lambda)} \\
\quad [(\text{’let (? symbol?) (var expr) \ldots)} \\
\quad \text{body \ldots)} \\
\quad (\text{format "named let, vars="s exprs="s" var expr})] \\
\quad [(\text{’let ((var expr) \ldots) body \ldots)} \\
\quad (\text{format "normal let, vars="s exprs="s" var expr})] \\
\quad (\text{format "malformed let"})]
\]

\text{(let-analyzer ’}(\text{let ((a b) (c d)) e f g})
⇒ "normal let, vars=(a c) exprs=(b d)"

(let-analyzer '(let foo ((x (f a b)) (y (f c d))) e f g))
⇒ "named let, vars=(x y) exprs=((f a b) (f c d))"

(let-analyzer '(let (a) b c d))
⇒ "malformed let"

Using = function application. The pattern variable \( m \) is matched to the result of application of the regular expression.

(match "gauche-ref\.texi"
  ((? string? (= #/(.*)\.([^.]+)/ \m))
   (format "base=\m 1 suffix=\m 2")
  )
⇒ "base=gauche-ref suffix=\m"

An example of quasipattern. In the first expression, the pattern except value is quoted, so the symbols the, answer, and is are not pattern variables but literal symbols. The second expression shows that; input symbol was does not match the literal symbol is in the pattern. If we don’t use quasiquote, all symbols in the pattern are pattern variables, so any four-element list matches as the third expression shows.

(match '(the answer is 42)
  ['(the answer is ,value) value]
  [_, #f])
⇒ 42

(match '(the answer was 42)
  ['(the answer is ,value) value]
  [_, #f])
⇒ #f

(match '(a b c d)
  [(the answer is value) value]
  [_, #f])
⇒ d

An example of matching records. The following code implements “rotation” operation to balance a red-black tree.

(define-record-type T #t #t
  color left value right)

(define (rbtree-rotate t)
  (match t
    [(or ($ T 'B ($ T 'R ($ T 'R a x b) y c) z d)
         ($ T 'B ($ T 'R a x ($ T 'R b y c)) z d)
         ($ T 'B a x ($ T 'R ($ T 'R b y c) z d))
         ($ T 'B a x ($ T 'R b y ($ T 'R c z d))))
     (make-T 'R (make-T 'B a x b) y (make-T 'B c z d))]
    [_, t]))

12.88 util.record - SLIB-compatible record type

util.record [Module]

This module provides a Guile and SLIB compatible record type API. It is built on top of Gauche's object system.
See also Section 9.28 [Record types], page 502, which provides a convenience macro `define-record-type`.

**make-record-type** `type-name` `field-names`  
{util.record} Returns a new class which represents a new record type. (It is what is called `record-type descriptor` in SLIB). In Gauche, the new class is a subclass of `record` (see Section 9.28 [Record types], page 502).

`type-name` is a string that is used for debugging purposes. It is converted to a symbol and set as the name of the new class. `field-names` is a list of symbols of the names of fields. Each field is implemented as a slot of the new class.

In the following procedures, `rtd` is the record class created by `make-record-type`.

**record-constructor** `rtd` :optional `field-names`  
{util.record} Returns a procedure that constructs an instance of the record type of given `rtd`. The returned procedure takes exactly as many arguments as `field-names`, which defaults to `'( )`. Each argument sets the initial value of the corresponding field in `field-names`.

**record-predicate** `rtd`  
{util.record} Returns a procedure that takes one argument, which returns `#t` iff the given argument is of type of `rtd`.

**record-accessor** `rtd` `field-name`  
{util.record} Returns an accessor procedure for the field named by `field-name` of type `rtd`. The accessor procedure takes an instance of `rtd`, and returns the value of the field.

**record-modifier** `rtd` `field-name`  
{util.record} Returns a modifier procedure for the field named by `field-name` of type `rtd`. The modifier procedure takes two arguments, an instance of `rtd` and a value, and sets the value to the specified field.

```
(define rtd (make-record-type "my-record" '(a b c)))

rtd ⇒ #<class my-record>

(define make-my-record (record-constructor rtd '(a b c)))

(define obj (make-my-record 1 2 3))

obj ⇒ #<my-record 0x819d9b0>

((record-predicate? rtd) obj) ⇒ #t

((record-accessor rtd 'a) obj) ⇒ 1
((record-accessor rtd 'b) obj) ⇒ 2
((record-accessor rtd 'c) obj) ⇒ 3

((record-modifier rtd 'a) obj -1)

((record-accessor rtd 'a) obj) ⇒ -1
```
12.89 util.relation - Relation framework

util.relation
Provide a set of common operations for relations.

Given set of values S1, S2, ..., Sn, a relation R is a set of tuples such that the first element of a tuple is from S1, the second from S2, ..., and the n-th from Sn. In another word, R is a subset of Cartesian product of S1, ..., Sn. (The definition, as well as the term relation, is taken from the Codd’s 1970 paper, “A Relational Model of Data for Large Shared Data Banks”, in CACM 13(6) pp.377–387.)

This definition can be applied to various datasets: A set of Gauche object system instances is a relation, if you view each instance as a tuple and each slot value as the actual values. A list of lists can be a relation. A stream that reads from CSV table produces a relation. Thus it would be useful to provide a module that implements generic operations on relations, no matter how the actual representation is.

From the operational point of view, we can treat any datastructure that provides the following four methods; relation-rows, which retrieves a collection of tuples (rows); relation-column-names, relation-accessor, and relation-modifier, which provide the means to access meta-information. All the rest of relational operations are built on top of those primitive methods.

A concrete implementation of relation can use duck typing, i.e. it doesn’t need to inherit a particular base class to use the relation methods. However, for the convenience, a base class <relation> is provided in this module. It works as a mixin class—a concrete class typically wants to inherit <relation> and <collection> or <sequence>. Check out the sample implementations in the lib/util/relation.scm in the source tree, if you’re curious.

This module is still under development. The plan is to build useful relational operations on top of the common methods.

Basic class and methods

<relation>
{util.relation} An abstract base class of relations.

relation-column-names (r <relation>)
{util.relation} A subclass must implement this method. It should return a sequence of names of the columns. The type of column names is up to the relation; we don’t place any restriction on it, as far as they are different each other in terms of equal?.

relation-accessor (r <relation>)
{util.relation} A subclass must implement this method. It should return a procedure that takes two arguments, a row from the relation r and a column name, and returns the value of the specified column.

relation-modifier (r <relation>)
{util.relation} A subclass must implement this method. It should returns a procedure that takes three arguments, a row from the relation r, a column name, and a value to set. If the relation is read-only, this method returns #f.

relation-rows (r <relation>)
{util.relation} A subclass must implement this method. It should return the underlying instance of <collection> or its subclass (e.g. <sequence>)

The rest of method are built on top of the above four methods. A subclass of <relation> may overload some of the methods below for better performance, though.
relation-column-name? (r <relation>) column
{util.relation} Returns true iff column is a valid column name for the relation r.

relation-column-getter (r <relation>) column
{util.relation} Returns a procedure to access the specified column of a row from the relation r. Relation-column-getter should return a procedure that takes one argument, a row. Relation-column-setter should return a procedure that takes two arguments, a row and a new value to set.

If the relation is read-only, relation-column-setter returns #f.

relation-column-setter (r <relation>) column
{util.relation} Returns a procedure to access the specified column of a row from the relation r. Relation-column-getter should return a procedure that takes one argument, a row.

relation-ref (r <relation>) row column :optional default
{util.relation} Row is a row from the relation r. Returns value of the column in row. If column is not a valid column name, default is returned if it is given, otherwise an error is signaled.

relation-set! (r <relation>) row column value
{util.relation} Row is a row from the relation r. Sets value as the value of column in row. This may signal an error if the relation is read-only.

relation-column-getters (r <relation>)
{util.relation} Returns full list of getters and setters. Usually the default method is sufficient, but the implementation may want to cache the list of getters, for example.

relation-column-setters (r <relation>)
{util.relation} Returns full list of getters and setters. Usually the default method is sufficient, but the implementation may want to cache the list of getters, for example.

relation-coercer (r <relation>)
{util.relation} Returns a procedure that coerces a row into a sequence. If the relation already uses a sequence to represent a row, it can return row as is.

relation-insertable? (r <relation>)
{util.relation} Returns true iff new rows can be inserted to the relation r.

relation-insert! (r <relation>) row
{util.relation} Insert a row row to the relation r.

relation-deletable? (r <relation>)
{util.relation} Returns true iff rows can be deleted from the relation r.

relation-delete! (r <relation>) row
{util.relation} Deletes a row row from the relation r.

relation-fold (r <relation>) proc seed column ...
{util.relation} Applies proc to the values of column ... of each row, passing seed as the state value. That is, for each row in r, proc is called as follows:

\[
\text{(proc } v_0 \; v_1 \ldots \; v_i \; \text{seed)}
\]

where \(v_k = (\text{relation-ref } r \; \text{row} \; \text{column}_k)\)

The result of the call becomes a new seed value, and the final result is returned from relation-fold.

For example, if a relation has a column named \text{amount}, and you want to sum up all of them in a relation \(r\), you can write like this:

\[
(\text{relation-fold } r \; + \; \text{‘amount})
\]
Concrete classes

<simple-relation>
{util.relation}

<object-set-relation>
{util.relation}

12.90 util.stream - Stream library

util.stream

This module provides a library of lazy streams, including the functions and syntaxes defined in SRFI-40 and SRFI-41, the latter of which became a part of R7RS large (as (scheme stream),
Gauche has a built-in lazy sequences (see Section 6.18.2 [Lazy sequences], page 241), which is a lazy stream integrated to a list so that you can use all list procedures on it. Lazy streams provided in this module are somewhat heavier than lazy sequences and you need to use special procedures, but it is strictly lazy (while a lazy sequence evaluates one element ahead) and portable.

12.90.1 Stream primitives

stream? obj

[Function]
[R7RS stream] {util.stream} Returns #t iff obj is a stream created by a procedure of util.stream.

stream-null

[Variable]
[R7RS stream] {util.stream} The singleton instance of NULL stream.

stream-cons object stream

[Macro]
[R7RS stream] {util.stream} A fundamental constructor of a stream. Adds object to the head of a stream, and returns a new stream.

stream-null? obj

[Function]
[R7RS stream] {util.stream} Returns #t iff obj is the null stream.

stream-pair? obj

[Function]
[R7RS stream] {util.stream} Returns #t iff obj is a non-null stream.

stream-car s

[Function]
[R7RS stream] {util.stream} Returns the first element of the stream s.

stream-cdr s

[Function]
[R7RS stream] {util.stream} Returns the remaining elements of the stream s, as a stream.

stream-delay expr

[Macro]
[SRFI-40]{util.stream} Returns a stream which is a delayed form of expr.

As a rule of thumb, any stream-producing functions should wrap the resulting expression by stream-delay. (Or you can use stream-lambda, stream-let or stream-define, described below.)

stream-lambda formals body body2 ...

[Macro]
[R7RS stream] {util.stream} A convenience macro to create a function that returns a stream. Effectively, (stream-lambda formals body body2 ...) is the same as (lambda formals (stream-delay body body2 ...)).
12.90.2 Stream constructors

stream obj ...

[Function]  {util.stream}  Returns a new stream whose elements are obj ... .  

Note: This differs from SRFI-41’s (scheme.stream’s) stream, which is a macro so that arguments are lazily evaluated. SRFI-41’s stream is provided as stream+ in this module.

(stream 1 2 3)  ⇒ a stream that contains (1 2 3)
(stream 1 (/ 1 0)))  ⇒ error

stream+ expr ...

{util.stream}  Returns a new stream whose elements are the result of expr ... .  

This is the same as SRFI-41(scheme.stream)’s stream. Each expr isn’t evaluated until it is accessed.

(define s (stream+ 1 (/ 1 0))) ;; doesn’t yield an error
(stream-car s)  ⇒ 1
(stream-cadr s)  ⇒ error

stream-unfold f p g seed

[Function]  {util.stream}  Creates a new stream whose element is determined as follows:

• A “go” predicate p is called on the current seed value. If it yields #f, the stream terminates.

• Otherwise, (f s) is the element of the stream, and (g s) becomes the next seed value, where s is the current seed value. The initial seed value is given by seed.

Note: Unfortunately, the order of arguments differs from other *-unfold procedures, which takes p f g (predicate, value-generator and seed-generator). Furthermore, the predicate is stop-predicate (returning true stops iteration).

(stream->list
  (stream-unfold integer->char (cut < <> 58) (cut + 1 <>) 48))
⇒ (#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)

stream-unfoldn f seed n

[Function]  {util.stream}  Creates n streams related each other, whose contents are generated by f and seed.

The f is called with the current seed value, and returns n+1 values:

(f seed)
⇒ seed result_0 result_1 ... result_n-1

The first value is to be the next seed value. Result_k must be one of the following forms:

(val)  val will be the next car of k-th stream.
#f  No new information for k-th stream.
()  The end of k-th stream has been reached.

The following example creates two streams, the first one produces an infinite series of odd numbers and the second produces evens.

  gosh> (define-values (s0 s1)
             (stream-unfoldn (lambda (i)
                             (values (+ i 2)))  ;; next seed
                             (list i))  ;; for the first stream
             ..., ...
(list (+ i 1))) ;; for the second stream

0 2)

#<undef>

(gosh> (stream->list (stream-take s0 10)))

(0 2 4 6 8 10 12 14 16 18)

(gosh> (stream->list (stream-take s1 10)))

(1 3 5 7 9 11 13 15 17 19)

stream-unfolds f seed

[Function]

[R7RS stream] {util.stream} Like stream-unfoldn, but the number of created streams is determined by the number of return values from f. See stream-unfoldn above for the details.

stream-constant obj ...

[Function]

[R7RS stream] {util.stream} Returns an infinite stream that repeats obj ....

(stream->list 10 (stream-constant 1 2))

⇒ (1 2 1 2 1 2 1 2 1 2)

make-stream n :optional init

{util.stream} Creates a new stream of n elements of init. If init is omitted, #f is used. Specifying a negative number to n creates an infinite stream.

stream-tabulate n init-proc

{util.stream} Creates a new stream of n elements. The k-th element is obtained by applying init-proc to k. Specifying a negative number to n creates an infinite stream.

stream-iota :optional count start step

{util.stream} Creates a new stream of numbers, starting from start and incrementing step. The length of stream is maximum integer not greater than nonnegative real number count. The default values of count, start and step are +inf.0, 0 and 1, respectively.

If start and step are exact, and count is exact or infinite, a sequence of exact numbers are created. Otherwise, a sequence of inexact numbers are created.

stream-range start :optional end step

[R7RS stream] {util.stream} Creates a new stream of real numbers, starting from start and stops before end, stepping by step. If end is omitted, positive infinity is assumed. If step is omitted, 1 is assumed if end is greater than start, and -1 if end is less than start.

The generated numbers are exact if start and step are exact and end is either exact or infinite. Otherwise, inexact numbers are generated.

In R7RS scheme.stream, end argument is required.

(stream->list (stream-range 0 10))

⇒ (0 1 2 3 4 5 6 7 8 9)

stream-from start :optional step

[R7RS stream] {util.stream} This is yet another number sequence generator. Generates an infinite sequence whose i-th element is (+ start (* i step)). If step is omitted, 1 is assumed. If both start and step are exact, exact numbers are generated. Otherwise, inexact numbers are generated.

stream-iterate f seed

[R7RS stream] {util.stream} Returns a stream starting from seed, and each successive element is calculated by (f s) where s is the previous element.

(stream->list 5 (stream-iterate (cut cons 'x <> ) '()))

⇒ (() (x) (x x) (x x x) (x x x x))

See also literate in gauche.lazy (see Section 9.15 [Lazy sequence utilities], page 451).
stream-xcons a b
{util.stream} (stream-cons b a). Just for convenience.

stream-cons* elt ... stream
{util.stream} Creates a new stream which appends elt ... before stream.

list->stream list
{R7RS stream} {util.stream} Returns a new stream whose elements are the elements in list.

string->stream string :optional tail-stream
{util.stream} Converts a string to a stream of characters. If an optional tail-stream is given, it becomes the tail of the resulting stream.

(string->stream "abc" (list->stream '(1 2 3))) ⇒ (#\a #\b #\c 1 2 3)

stream-format fmt arg ...
{util.stream} Returns a stream which is a result of applying string->stream to (format fmt arg ...).

port->stream :optional iport reader closer
{R7RS stream} {util.stream} Creates a stream, whose elements consist of the items read from the input port iport. The default iport is the current input port. The default reader is read-char.

The result stream terminates at the point where reader returns EOF (EOF itself is not included in the stream). The port won’t be closed by default when it reaches EOF.

If closer is given, it is called with iport as an argument just after reader reads EOF. You can close the port with it.

The reader and closer arguments are Gauche’s extension. R7RS scheme.stream only takes one optional argument, iport.

generator->stream gen
{util.stream} Creates a lazy stream of values generated by a generator gen. A generator is a thunk that returns a value every time it is called, with returning EOF to indicate the end of the input. See Section 9.12 [Generators], page 436, for the details of generators.

See also generator->lseq, which is another way to get a lazy sequence from a generator (see Section 6.18.2 [Lazy sequences], page 241, for the details).

iterator->stream iter
{util.stream} A generic procedure to turn an internal iterator iter into a stream of iterated results.

The iter argument is a procedure that takes two arguments, next and end, where next is a procedure that takes one argument and end is a thunk. iter is supposed to iterate over some set and call next for each argument, then call end to indicate the end of the iteration. Here’s a contrived example:

(stream->list
 (iterator->stream
 (lambda (next end) (for-each next '(1 2 3 4 5)) (end)))
)⇒ (1 2 3 4 5)

Internally iterator->stream uses the “inversion of iterator” technique, so that iter only iterates to the element that are needed by the stream. Thus iter can iterate over an infinite set. In the following example, iter is an infinite loop calling next with increasing integers, but only the first 10 elements are calculated because of stream-take:

(stream->list
(stream-take
 (iterator->stream
  (lambda (next end)
   (let loop ((n 0)) (next n) (loop (+ n 1)))))
10))
⇒ (0 1 2 3 4 5 6 7 8 9)

stream-of elt-expr clause ...  [Macro]
[R7RS stream] {util.stream} Stream comprehension. Returns a stream in which each
element is computed by elt-expr. The clause creates scope of elt-expr and controls iterations.
Each clause can be one of the following forms:

(x in stream-expr)
  Iterate over stream-expr, binding x to each element in each iteration. The variable x is visible in successive clauses and elt-expr

(x is expr)
  Bind a variable x with the value of expr. The variable x is visible in successive clauses and elt-expr.

expr
  If expr evaluates to #f, this iteration is skipped without generating a new element.

The following comprehension generates infinite sequence of pythagorean triples:

(define pythagorean-triples
 (stream-of (list a b c)
   (c in (stream-from 3))
   (b in (stream-range 2 c))
   (a in (stream-range 1 b))
   (= (square c) (+ (square b) (square a)))))

(stream->list 5 pythagorean-triples)
⇒ ((3 4 5) (6 8 10) (5 12 13) (9 12 15) (8 15 17))

12.90.3 Stream binding

define-stream (name . formals) body body2 . . .  [Macro]
[R7RS stream] {util.stream} A convenient macro to define a procedure that yields a stream.
Same as the following form:

(define (name . formals)
 (stream-delay
  (let () body body2 ...)))

stream-let loop-var ([var init] ...) body body2 . . .  [Macro]
[R7RS stream] {util.stream} A handy macro to write a lazy named-let loop. It is the same
as the following:

(let loop-var ([var init] ...)
 (stream-delay
  (let () body body2 ...)))

stream-match stream-expr clause ...  [Macro]
[R7RS stream] {util.stream} This allows accessing streams via simple pattern matching.
The stream-expr argument is evaluated and must yield a stream. Each clause must be either
a form of (pattern expr) or (pattern fender expr).
The content of the stream is matched to each pattern, which must have one of the following
forms:

()  Matches a null stream.
Matches a stream that has exactly the same number of elements as the number of pattern elements.

\((p_0 \; p_1 \ldots)\)

Matches a stream that has at least the same number of elements as the number of pattern elements except \(p_{Rest}\). The rest of stream matches with \(p_{Rest}\).

\(p_{Rest}\)

Matches an entire stream.

Each pattern element can be an identifier or a literal underscore. If it is an identifier, it is bound to the matched element while evaluating the corresponding \(fender\) and \(expr\).

If \(fender\) is present in the \(clause\), it is evaluated; if it yields \#f, the match of the \(clause\) fails and next clauses will be tried.

Otherwise, \(expr\) is evaluated and the result(s) becomes the result(s) of \(stream\)-\(match\).

Only the elements from the stream that is required to match are accessed.

The following example defines a procedure to count the number of true values in the stream:

\[
\begin{align*}
&\text{(define (num-trues strm)} \\
&(\text{stream-match strm} \\
&\quad ((\text{} 0) \\
&\quad \quad ((\text{head . tail}) \text{ head (+ 1 (num-trues tail))}) \\
&\quad \quad ((\_ . tail) \text{ (num-trues tail)))}) \\
&\quad (\text{num-trues (stream #f #f #t #f #t #f #t #t)})) \\
&\Rightarrow 4
\end{align*}
\]

### 12.90.4 Stream consumers

These procedures takes stream(s) and consumes its/their elements until one of the streams is exhausted.

\textbf{stream-for-each func . streams}

[Function]

[R7RS stream] \{util.stream\} Applies \(func\) for each element of \(streams\). Terminates if one of \(streams\) reaches the end.

\textbf{stream-fold f seed stream}

[Function]

[R7RS stream] \{util.stream\} Apply \(f\) on the current seed value and an element from \(stream\) to obtain the next seed value, and repeat it until \(stream\) is exhausted, then returns the last seed value. The initial seed value is given by \(seed\).

Note: The argument order of \(f\) differs from other *-fold procedures, e.g. \(fold\) (see Section 6.6.6 [Walking over lists], page 147) takes the element first, then the seed value.

\[
\text{(stream-fold - 0 (stream 1 2 3 4 5))} \\
\Rightarrow -15
\]

### 12.90.5 Stream operations

\textbf{stream-append stream ...}

[Function]

[R7RS stream] \{util.stream\} Returns a new stream which is concatenation of given streams.

\textbf{stream-concat streams}

[Function]

\textbf{stream-concatenate streams}

[Function]

[R7RS stream] \{util.stream\} R7RS scheme.stream defines \textbf{stream-concat}. Gauche had \textbf{stream-concatenate}, and keeps it for the backward compatibility. Both are the same.

The \(streams\) argument is a stream of streams. Returns a new stream that is concatenation of them. Unlike \textbf{stream-append}, \(streams\) can generate infinite streams.
**stream-map** `func stream stream2 ...`  
[Function]  
[R7RS stream] {util.stream} Returns a new stream, whose elements are calculated by applying `func` to each element of `stream` ... .

**stream-scan** `func seed stream`  
[Function]  
[R7RS stream] {util.stream} Returns a stream of `seed`, `(func seed e0)`, `(func (func seed e0) e1)`, ... , where `e0`, `e1` ... are the elements from the input `stream`. If `stream` is finite, the result stream has one more elements than the number of elements in the original stream.

```scheme
(stream->list
 (stream-scan xcons () (stream 1 2 3 4 5)))
⇒ ((1) (1) (2 1) (3 2 1) (4 3 2 1) (5 4 3 2 1))
```

**stream-zip** `stream ...`  
[Function]  
[R7RS stream] {util.stream} Returns a new stream whose elements are lists of corresponding elements from input `streams`. The output stream ends when one of input streams is exhausted.

```scheme
(stream->list
 (stream-zip (stream 1 2 3) (stream 'a 'b 'c 'd)))
⇒ ((1 a) (2 b) (3 c))
```

**stream-filter** `pred stream`  
[Function]  
[R7RS stream] {util.stream} Returns a new stream including only elements passing `pred`. This procedure returns immediately; `stream` can be infinite.

**stream-remove** `pred stream`  
[Function]  
{util.stream} Returns a new stream including only elements that doesn’t satisfy `pred`. It is the same as `(stream-filter (complement pred) stream)`.

**stream-partition** `pred stream`  
[Function]  
{util.stream} Returns two streams, one consists of the elements in `stream` that satisfy `pred`, the other consists of the ones that doesn’t satisfy `pred`.

**stream->list** `stream`  
[Function]  
[R7RS stream] {util.stream} Converts a stream to a list. In the first form, all elements from `stream` are taken (thus it never returns if `stream` is infinite). In the second form, at most `n` elements are taken, where `n` must be a nonnegative exact integer.

Note: In usual Scheme conventions, the optional `n` comes after the main argument (`stream`).

**stream->string** `stream`  
[Function]  
{util.stream} Converts a stream to a string. All elements of `stream` must be characters, or an error is signaled.

**stream-lines** `stream`  
[Function]  
{util.stream} Splits `stream` where its element equals to `#
`, and returns a stream of splitted streams. The character `#
` won’t be included in the results.

```scheme
(stream->list
 (stream-map stream->string
 (stream-lines (string->stream "abc
def
ghi"))))
⇒ ("abc" "def" "ghi")
```

**stream=** `elt= stream ...`  
[Function]  
{util.stream} Returns true iff each corresponding element of `stream ...` are the same in terms of `elt=`, which takes two arguments. This procedure won’t terminate if all of `streams` is infinite.
stream-prefix= stream prefix :optional elt=  
{util.stream} Compares initial elements of stream against a list prefix by elt=. Returns true iff they match. Only as many elements of stream as prefix has are checked.

stream-caar s  
stream-cadr s  
...  
stream-cdddar s  
stream-cddddr s  
{util.stream} (stream-caar s) = (stream-car (stream-car s)) etc.

stream-ref stream pos  
{R7RS stream} {util.stream} Returns the pos-th element in the stream. Pos must be a nonnegative exact integer.

stream-first s  
stream-second s  
stream-third s  
stream-fourth s  
stream-fifth s  
stream-sixth s  
stream-seventh s  
stream-eighth s  
stream-ninth s  
stream-tenth s  
{util.stream} (stream-first s) = (stream-ref s 0) etc.

stream-take stream count  
stream-take-safe stream count  
{util.stream} Returns a new stream that consists of the first count elements of the given stream. If the given stream has less than count elements, the stream returned by stream-take would raise an error when the elements beyond the original stream is accessed. On the other hand, the stream returned by stream-take-safe will return a shortened stream when the given stream has less than count elements.

(stream->list (stream-take (stream-iota -1) 10))  
⇒ (0 1 2 3 4 5 6 7 8 9)

(stream-take (stream 1 2) 5)  
⇒ stream

(stream->list (stream-take (stream 1 2) 5))  
⇒ error

(stream->list (stream-take-safe (stream 1 2) 5))  
⇒ (1 2)

Note: SRFI-41 (scheme.stream) also defines stream-take, but the argument order is reversed, and also it allows stream to have less than count elements.

stream-drop stream count  
stream-drop-safe stream count  
{util.stream} Returns a new stream that consists of the elements in the given stream except the first count elements. If the given stream has less than count elements, stream-drop...
returns a stream that raises an error if its element is accessed, and \texttt{stream-drop-safe} returns an empty stream.

Note: SRFI-41 (\texttt{scheme.stream}) also defines \texttt{stream-drop}, but the argument order is reversed, and also it allows \texttt{stream} to have less than \texttt{count} elements.

\textbf{stream-intersperse stream element} \hfill [Function]
\{util.stream\} Returns a new stream in which \texttt{element} is inserted between each consecutive two elements of \texttt{stream}.

\[(\text{stream->list (stream-intersperse (stream 1 2 3) 0))} \Rightarrow (1 \ 0 \ 2 \ 0 \ 3)\]

\textbf{stream-split stream pred} \hfill [Function]
\{util.stream\} Split \texttt{stream} on elements that satisfy \texttt{pred}, and returns a stream of splitted streams. The delimiting element won’t be included in the result.

\[(\text{stream->list (stream-map stream->list (stream-split (stream 1 3 2 5 7 9 6 8 1) even?)))} \Rightarrow ((1 \ 3) (5 \ 7 \ 9) () (1))\]

\textbf{stream-last stream} \hfill [Function]
\{util.stream\} Returns the last element of \texttt{stream}. If \texttt{stream} is empty, an error is thrown. If \texttt{stream} is infinite, this procedure won’t return.

\textbf{stream-last-n stream count} \hfill [Function]
\{util.stream\} Returns a stream that contains the last \texttt{count} elements in \texttt{stream}. The \texttt{count} argument must be a nonnegative exact integer. If the length of \texttt{stream} is smaller than \texttt{count}, the resulting stream is the same as \texttt{stream}. If \texttt{stream} is infinite, this procedure won’t return.

\textbf{stream-butlast stream} \hfill [Function]
\{util.stream\} Returns a stream which is the same as \texttt{stream} but without the last element. If \texttt{stream} is empty, an empty stream is returned. immediately. The argument can be an infinite stream.

\textbf{stream-butlast-n stream count} \hfill [Function]
\{util.stream\} Returns a stream which is the same as \texttt{stream} but without the last \texttt{count} elements. The \texttt{count} argument must be a nonnegative exact integer. If \texttt{stream} is shorter than \texttt{count}, a null stream is returned. \texttt{Stream} can be infinite.

\textbf{stream-length stream} \hfill [Function]
[R7RS stream] \{util.stream\} Returns the number of elements in \texttt{stream}. It diverges if \texttt{stream} is infinite.

\textbf{stream-length>= stream n} \hfill [Function]
\textbf{stream-length= stream n} \hfill [Function]
\{util.stream\} Returns true iff the length of \texttt{stream} is greater than or equal to, and exactly equal to, \texttt{n}, respectively. These procedures only realizes \texttt{stream} up to \texttt{n} elements.

\textbf{stream-reverse stream \optional tail-stream} \hfill [Function]
[R7RS stream] \{util.stream\} Returns a stream that returns the elements of \texttt{stream} in reverse order. If \texttt{tail-stream} is given, it is added after the reversed stream.

The \texttt{optional} argument is Gauche’s extension. The description of \texttt{reverse} (see Section 6.6.7 [Other list procedures], page 151) for more details.
Chapter 12: Library modules - Utilities

**stream-count** pred stream ...  
{util.stream} The predicate pred is called with each of the first element of stream . . ., then each of the second element, and so on, until any of the stream is exhausted. Then returns the number that pred returns a true value. At least one stream needs to be finite.

\[
\text{(stream-count < (stream 1 2 3 4 5) (stream 2 2 2 10 10))} \\
\Rightarrow 3
\]

**stream-find** pred stream  
{util.stream} The predicate pred is applied to each element of stream in turn, and as soon as the pred returns true, returns the element that caused pred to return true. Remaining elements of stream won’t be realized.

If no elements in stream satisfy pred, #f is returned.

**stream-find-tail** pred stream  
{util.stream} The predicate pred is applied to each element of stream in turn, and as soon as the pred returns true, returns a stream that contains the element and the rest of the input stream. Remaining elements of stream won’t be realized.

If no elements in stream satisfy pred, the empty stream is returned.

**stream-take-while** pred stream  
**stream-drop-while** pred stream  
{R7RS stream} {util.stream} Returns a stream that contains or excludes initial consecutive elements of stream that satisfy pred, respectively.

\[
\text{(stream->list (stream-take-while even? (stream 2 4 6 1 4 2)))} \\
\Rightarrow (2 4 6) \\
\text{(stream->list (stream-drop-while even? (stream 2 4 6 1 4 2)))} \\
\Rightarrow (1 4 2)
\]

**stream-span** pred stream  
**stream-break** pred stream  
{util.stream} Return two streams; the first one consists of the initial consequent elements of stream that satisfy or dissatisfy pred, and the second one is the rest of the stream.

\[
\text{(map stream->list} \\
\text{(values->list (stream-span even? (stream 2 4 6 1 4 2))))} \\
\Rightarrow ((2 4 6) (1 4 2)) \\
\text{(map stream->list} \\
\text{(values->list (stream-break even? (stream 1 9 6 1 4 2))))} \\
\Rightarrow (((1 9) (6 1 4 2)))
\]

**stream-any** pred stream ...  
**stream-every** pred stream ...  
{util.stream} The predicate pred is applied to each of the first elements of stream . . . , then second, and so on. When pred returns a true value, stream-any returns the value immediately without looking further. When pred returns #f, stream-every returns #f immediately without looking further.

If none of applications of pred returns a true value until one of the streams is exhausted, stream-any returns #f. If none of applications of pred returns #f until one of the streams is exhausted, stream-every returns the last result of pred.

\[
\text{(stream-any } [^x y] \text{ (and (> x y) (list x y)))} \\
\text{(stream 1 2 3 4 5)} \\
\text{(stream 3 3 3 3 3)} \\
\Rightarrow (4 3)
\]
(stream-every (^[x y] (and (> x y) (list x y)))
  (stream 6 6 6 6)
  (stream 1 2 3 4 5))
  ⇒  (6 5)

stream-index pred stream . . .
{util.stream} The predicate pred is applied to each of the first elements of stream . . . , then second, and so on. If it returns a true value, stream-index returns immediately with the index. If pred never returns a true value until one of the streams is exhausted, stream-index returns #f.

  (stream-index odd? (stream 2 4 6 7 8))
  ⇒  3

stream-member obj stream :optional elt=
{Function}
stream-memq obj stream
{Function}
stream-memv obj stream
{Function}
{util.stream} Stream version of member, memq, and memv (see Section 6.6.7 [Other list procedures], page 151). Search an element equivalent to obj from stream, and if it is found, returns a stream consists of the element and the rest of the stream. If no such element is found, #f is returned.

They differ regarding which equivalent procedure to be used; stream-member uses the given elt= procedure, or equal? if omitted, stream-memq uses eq?, and stream-memv uses eqv?.

stream-delete obj stream :optional elt=
{util.stream} Equivalent to (stream-remove (cut elt= obj <>)) stream), that is, returns a stream that is the same as input except objs. If elt= is omitted, equal? is used.

stream-delete-duplicates stream :optional elt=
{util.stream} Returns a stream that is the same as input except the consecutive equivalent elements are removed but the first one. Elements are compared with elt=, defaulted to equal?.

stream-grep re stream
{util.stream} The re argument must be a regexp object, or a string representation of regexp (which is converted to regexp by string->regexp).

Returns a stream that contains input elements that matches re.

write-stream stream :optional oport writer
{util.stream} Writes each element of stream to oport, using writer. When oport is omitted, the current output port is used. When writer is omitted, write-char is used.

No 'spacer' is written between elements. As the default suggests, this is most useful to write out a stream of characters.

12.91 util.temporal-relation - Temporal relation

util.temporal-relation
{Module}
Procedures to find relation between two temporal intervals, or a temporal interval and a point in time.

A temporal interval is represented by two points in time, lesser bound and greater bound. This module does not define a concrete structure for temporal intervals. Instead, the user provides a interval protocol, which specifies how to access start/end time and how to compare them.
Given two temporal intervals x and y, there are 13 possible relations (the terms are taken from Haskell Rampart library https://hackage.haskell.org/package/rampart-2.0.0.0/docs/Rampart.html):

- x before y
- x meets y
- x overlaps y
- x finished-by y
- x contains y
- x starts y
- x equal y
- x started-by y
- x during y
- x finishes y
- x overlapped-by y
- x met-by y
- x after y

make-interval-protocol lesser greater :optional compare-points [Function]
{util.temporal-relation} Creates and returns an interval protocol. The lesser and greater arguments are procedures taking an user-defined interval object, and returns its lesser bound and greater bound, respectively. The compare-points argument is a procedure that takes two points of time, and must return either a negative real value (if the first point is before the second), zero (if two points are the same), or a positive real value (if the first point is after the second).

The actual representation of time points are up to the caller.
If the compare-points argument is omitted, the built-in compare procedure is used (see Section 6.2.2 [Comparison], page 112). It is suffice for most scalar values; e.g. you may use real numbers, <time>, or <date> objects.

**pair-interval-protocol**  
{util.temporal-relation} Bound to `(make-interval-protocol car cdr)`; that is, an interval represented by cons of lesser bound and greater bound.

**relation? obj**  
{util.temporal-relation} Returns `#t` iff obj is a symbol representing a temporal relation. Valid symbols are one of before, meets, overlaps, finished-by, contains, starts, equal, started-by, during, finishes, overlapped-by, met-by and after.

**inverse rel**  
{util.temporal-relation} The argument must be a symbol representing a temporal relation (see relation? above for the valid symbols). This procedure returns its inverse relation; that is, `(relate _ x y) ≡ (inverse (relate _ y x))`, where _ is an interval protocol. An error is thrown if rel is not a valid relation.

**relate proto x y**  
{util.temporal-relation} The proto argument is an interval protocol, and both x and y must be temporal intervals that is compatible to the protocol proto.

The procedure returns a temporal relation, represented as one of the following symbols: before, meets, overlaps, finished-by, contains, starts, equal, started-by, during, finishes, overlapped-by, met-by and after.

**relate-point proto x point**  
{util.temporal-relation} The proto argument is an interval protocol, and x must be a temporal interval that implements the protocol. The point is a point of time, comparable by the compare-points procedure of the protocol.

The possible relations are a subset of interval-interval relations:

```
|<-- x -->|  
p       x before p
|<--- x --->|  
p       x finished-by p
|<---- x ---->|  
p       x contains p
|<---- x ---->|  
p       x started-by p
|<-- x -->|  
p       x after p
```

### 12.92 util.toposort - Topological sort

**util.toposort**  
Implements topological sort algorithm.
topological-sort graph :optional eqproc
{util.toposort} Graph represents a directed acyclic graph (DAG) by a list of connections, where each connection is the form
  (<node> <downstream> <downstream2> ...)
that means a node <node> is connected to other nodes <downstream> etc. <node> can be arbitrary object, as far as it can be compared by the procedure eqproc, which is eqv? by default (see Section 6.2.1 [Equality], page 110). Returns a list of <node>s sorted topologically. If the graph contains circular reference, an error is signaled.

12.93 util.unification - Unification

util.unification
{Module}

Implements unification algorithm.

The base API operates on abstract trees, while it is agnostic to the actual representation of the tree. The caller passes comparators and operators along the trees to unify.

We assume the abstract tree has the following structure:
  Tree : Variable | Value | Tuple
  Tuple : { Tree ... }
Here, {...} just represents a sequence of trees.

A variable can be bound to a tree. A value can only match itself.

To operate on this tree, we need the following comparators and procedures, which the API takes as arguments:

Variable comparator: var-cmpr
  A comparator to see if an item is a variable, and also check equality of two variables. It must be hashable. See Section 6.2.4 [Basic comparators], page 116, for the details of comparators.

Value comparator: val-cmpr
  A comparator to see if an item is a value, and also check equality of two values. Note that if a tree satisfies neither var-cmpr nor val-compr, it is regarded as a tuple.

Tuple folder: tuple-fold
  A procedure (tuple-fold job seed tuple1 [tuple2]). This procedure should work like fold (see Section 6.6.6 [Walking over lists], page 147) over the elements in the tuple(s). It is only called with either one or two tuples.

Tuple constructor: make-tuple
  A procedure (make-tuple proto elements), where proto is a tuple and elements are a list of trees. It must return a new tuple with the given elements, while all other properties are the same as proto. This procedure isn’t needed by unify.

unify a b var-cmpr val-cmpr tuple-fold
{util.unification} Unify two trees a and b and returns a substitution dictionary, which is a dictionary that maps variables to its bounded trees. See the entry of util.unification above for the description of var-cmpr, val-cmpr and tuple-fold.

  (dict->alist (unify '(a 3 (c b)) '(c b (2 e))
       symbol-comparator
       number-comparator
       fold))
⇒ ((e . 3) (a . c) (b . 3) (c . 2))

As you can see in the example above, a variable may be mapped to another variable, or even to a tree that contains variables. If you apply the substitution to the original tree, you must do it recursively until all the variables in the dictionary is eliminated.

If two trees cannot be unified, #f is returned.

```
(unify '(a (a)) '(x x) symbol-comparator number-comparator fold)
⇒ #f
```

#### unify-merge

```
unify-merge a b var-cmpr val-cmpr tuple-fold make-tuple
```

{util.unification} Unify two trees a and b, and apply the result substitutions to create a new tree eliminating variables.

See the entry of util.unification above for the description of var-cmpr, val-cmpr, tuple-fold and make-tuple.

```
(unify-merge '(a 3 (c b)) '(c b (2 e))
  symbol-comparator
  number-comparator
  fold
  ("^[_ elts] elts))
⇒ (2 3 (2 3))
```

If two trees can’t be unified, #f is returned.

### 12.94 www.cgi - CGI utility

#### www.cgi

Provides a few basic functions useful to write a CGI script.

In order to write CGI script easily, you may want to use other modules, such as rfc.uri (see Section 12.55 [URI parsing and construction], page 949), text.html-lite (see Section 12.70 [Simple HTML document construction], page 1002) and text.tree (see Section 12.80 [Lazy text construction], page 1014).

Note: it seems that there is no active formal specification for CGI. See http://w3c.org/CGI/ for more information.

#### Metavariables

**cgi-metavariables** :optional metavariables

{www.cgi} Normally, httpd passes a cgi program various information via environment variables. Most procedures in www.cgi refer to them (meta-variables). However, it is sometimes inconvenient to require environment variable access while you’re developing cgi-related programs. With this parameter, you can overrides the information of meta-variables.

Metavariables should be a list of two-element lists. Car of each inner list names the variable, and its cadr gives the value of the variable by string.

For example, the following code overrides REQUEST_METHOD and QUERY_STRING meta-variables during execution of my-cgi-procedure. (See Section 6.16.1 [Parameters], page 233, for the details of parameterize).

```
(parameterize ((cgi-metavariables '(("REQUEST_METHOD" "GET")
                                         ("QUERY_STRING" "x=foo"))))

      (my-cgi-procedure))
```

**cgi-get-metavariable name**

{www.cgi} Returns a value of cgi metavariable name. This function first searches the parameter cgi-metavariables, and if the named variable is not found, calls sys-getenv.
CGI scripts may want to use `cgi-get-metavariable` instead of directly calling `sys-getenv`; doing so makes reuse of the script easier.

### Parameter extraction

`cgi-parse-parameters :key :query-string :merge-cookies :part-handlers`  
{www.cgi} Parses query string and returns associative list of parameters. When a keyword argument `query-string` is given, it is used as a source query string. Otherwise, the function checks the metavariable `REQUEST_METHOD` and obtain the query string depending on the value (either from stdin or from the metavariable `QUERY_STRING`). If such a metavariable is not defined and the current input port is a terminal, the function prompts the user to type parameters; it is useful for interactive debugging.

If `REQUEST_METHOD` is `POST`, this procedure can handle both `application/x-www-form-urlencoded` and `multipart/form-data` as the enctype. The latter is usually used if the form has file-uploading capability.

When the post data is sent by `multipart/form-data`, each content of the part is treated as a value of the parameter. That is, the content of uploaded file will be seen as one big chunk of a string. The other information, such as the original file name, is discarded. If it is not desirable to read entire file into a string, you can customize the behavior by the `part-handler` argument. The details are explained in the "Handling file uploads" section below.

When a true value is given to `merge-cookies`, the cookie values obtained from the metavariable `HTTP_COOKIE` are appended to the result.

Note that the query parameter may have multiple values, so `cdr` of each element in the result is a list, not an atom. If no value is given to the parameter, `#t` is placed as its value. See the following example:

```lisp
(cgi-parse-parameters
  :query-string "foo=123&bar=%22%3f%3f%22&bar=zz&buzz")
⇒ (("foo" "123") ("bar "??" "zz") ("buzz" #t))
```

### Output generation

`cgi-header :key status content-type location cookies`  
{www.cgi} Creates a text tree (see Section 12.80 [Lazy text construction], page 1014) for the HTTP header of the reply message. The most simple form is like this:

```lisp
(tree->string (cgi-header))
⇒ "Content-type: text/html\r\n\n"
```

You can specify alternative content-type by the keyword argument `content-type`. If you want to set cookies to the client, specify a list of cookie strings to the keyword argument.
cookies. You can use `construct-cookie-string` (see Section 12.43 [HTTP cookie handling], page 924) to build such a list of cookie strings.

The keyword argument `location` may be used to generate a `Location:` header to redirect the client to the specified URI. You can also specify the `Status:` header by the keyword argument `status`. A typical way to redirect the client is as follows:

```scheme
(cgi-header :status "302 Moved Temporarily"
 :location target-uri)
```

### Convenience procedures

#### cgi-main

**Function**

```scheme
{www.cgi} A convenient wrapper function for CGI script. This function calls `cgi-parse-parameters`, then calls `proc` with the result of `cgi-parse-parameters`. The keyword argument `merge-cookies is passed to `cgi-parse-parameters`. `proc` has to return a tree of strings (see Section 12.80 [Lazy text construction], page 1014), including the HTTP header. `cgi-main` outputs the returned tree to the current output port by `write-tree`, then returns zero.

If an error is signaled in `proc`, it is caught and an HTML page reporting the error is generated. You can customize the error page by providing a procedure to the `on-error` keyword argument. The procedure takes an `<condition>` object (see Section 6.19.4 [Conditions], page 253), and has to return a tree of string for the error reporting HTML page, including an HTTP header.

When output the result, `cgi-main` refers to the value of the parameter `cgi-output-character-encoding`, and converts the character encoding if necessary.

The output behavior of `cgi-main` can be customized by a keyword argument `output-proc`; if it is given, the text tree (either the normal return value of `proc`, or an error page constructed by the error handler) is passed to the procedure given to `output-proc`. The procedure is responsible to format and output a text to the current output port, including character conversions, if necessary.

The keyword argument `part-handlers` are simply passed to `cgi-parse-parameters`, by which you can customize how the file uploads should be handled. See the "Handling file uploads" section below for the details.

If you specify to use temporary file(s) by it, `cgi-main` makes sure to clean up them whenever `proc` exits, even by error. See `cgi-add-temporary-file` below to utilize this feature for other purpose.

Before calling `proc`, `cgi-main` changes the buffering mode of the current error port to `:line` (See `port-buffering` in Section 6.21.3 [Common port operations], page 260, for the details about the buffering mode). This makes the error output easier for web servers to capture.

The following example shows the parameters given to the CGI program.

```scheme
#!/usr/local/bin/gosh

(use text.html-lite)
(use www.cgi)

(define (main args)

)
(cgi-main
 (lambda (params)
 '('
,(cgi-header)
, (html-doctype)
, (html:html
  (html:head (html:title "Example")))
(html:body
  (html:table :border 1
    (html:tr (html:th "Name") (html:th "Value"))
    (map (lambda (p)
       (html:tr
        (html:td (html-escape-string (car p)))
        (html:td (html-escape-string (x->string (cdr p))))
        (html-escape-string (x->string (cdr p)))))))
))))

cgi-add-temporary-file filename  [Function]
{www.cgi} This is supposed to be called inside proc of cgi-main. It registers filename as a temporary file, which should be unlinked when proc exits. It is a convenient way to ensure that your cgi script won’t leave garbages even if it throws an error. It is OK in proc to unlink or rename filename after calling this procedure.

cgi-temporary-files  [Parameter]
{www.cgi} Keeps a list of filenames registered by cgi-add-temporary-file.

Handling file uploads

As explained in cgi-parse-parameters above, file uploads are handled transparently by default, taking the file content as the value of the parameter. Sometimes you might want to change this behavior, for the file might be quite big and you don’t want to keep around a huge chunk of a string in memory. It is possible to customize handling of file uploads of cgi-parse-parameters and cgi-main by part-handlers argument. (The argument is only effective for the form data submitted by multipart/form-data enctype)

The part-handlers argument is, if given, a list of lists; each inner list is a form of (name-pattern action kv-list ...). Each uploaded file with a matching parameter name with name-pattern is handled according to action. (Here, a parameter name is the 'name' attribute given to the input element in the submitted form, not the name of the uploaded file).

Name-pattern must be either a list of string (matches one of them), a regexp, or #t (matches anything).

Action must be either one of the followings:

#f Default action, i.e. the content of the uploaded file is turned into a string and becomes the value of the parameter.

ignore The uploaded content is discarded.

file The uploaded content is saved in a temporary file. The value of the parameter is the pathname of the temporary file.

For this action, you can write an entry like (name-pattern file prefix), to specify the prefix of the pathname of the temporary file. For example, if you specify ("image" file "/var/mycgi/incoming/img"), the file uploaded as "image" parameter will be stored as something like /var/mycgi/incoming/img49g2Ua.
The application should move the temporary file to appropriate location; if you’re using `cgi-main`, the temporary files created by this action will be unlinked when `cgi-main` exits.

**file+name**

Like `file`, but the value of the parameter is a list of temporary filename and the filename passed by the client. It is useful if you want to use client’s filename (but do not blindly assume the client sends a valid pathname; for example, you shouldn’t use it to rename the uploaded file without validating it).

**procedure**

In this case, `procedure` is called to handle the uploaded contents. It is called with four arguments: `(procedure name filename part-info iport)`.

- `Name` is the name of the parameter.
- `Filename` is the name of the original file (pathname in the client).
- `Part-info` is a `<mime-part>` object that keeps information about this mime part, and `iport` is where the body can be read from.

For the details about these arguments, see Section 12.51 [MIME message handling], page 939; you might be able to use procedures provided by `rfc.mime`, such as `mime-retrieve-body`, to construct your own procedure.

If you create a temporary file in `procedure`, you can call `cgi-add-temporary-file` to make sure it is removed even if an error occurs during `cgi` processing.

If `kv-list` is given after `action`, it must be a keyword-value list and further modifies `action`. The following keywords are supported.

- **:prefix** Valid only if `action` is either `file` or `file+name`. Specifies the prefix of the temporary file. If you give `:prefix "/tmp/foo"`, for example, the file is saved as something like `/tmp/fooxAgjeQ`.
- **:mode** Valid only if `action` is either `file` or `file+name`. Specifies the mode of the temporary file in unix-style integer. By default it is `#o600`.

Note that the parameters that are not file uploads are not the subject of `part-handlers`; such parameter values are always turned into a string.

Here’s a short example. Suppose you have a form like this:

```html
<form enctype="multipart/form-data" method="POST" action="mycgi.cgi">
  <input type="file" name="imagefile" />
  <input type="text" name="description" />
  <input type="hidden" name="mode" value="normal" />
</form>
```

If you use `cgi-parse-parameters` in `mycgi.cgi` without `part-handlers` argument, you’ll get something like the following as the result. (The actual values depend on how the web client filled the form).

```lisp
(\"imagefile\" #*".....(image file content as a string)....")
\"description\" "my image")
\"mode\" "normal")
```

If you pass `\"imagefile\" file :prefix "/tmp/mycgi\"`) to `part-handlers` instead, you might get something like the following, with the content of uploaded file saved in `/tmp/mycgi7gq0B

```lisp
\"imagefile\" "/tmp/mycgi7gq0B")
\"description\" "my image")
\"mode\" "normal")
```

If you use a symbol `file+name` instead of `file` above, you’ll get something like `("/tmp/mycgi7gq0B" "logo.jpg")` as the value of `"imagefile"`, where `"logo.jpg"` is the
client-side filename. (Note: the client can send any string as the name of the file, so never assume it is a valid pathname).

### 12.95 www.cgi.test - CGI testing

**www.cgi.test**

This module defines a useful procedures to test CGI script. The test actually runs the named script, with specified environment variable settings, and retrieve the output. Your test procedure then examine whether the output is as expected or not.

**cgi-test-environment-ref**

<table>
<thead>
<tr>
<th>envvar-name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>(setter) cgitest-environment-ref</td>
<td>envvar-name value</td>
</tr>
</tbody>
</table>

{www.cgi.test} The module keeps a table of default values of environment variables with which the cgi script will be run. These procedures allow the programmer to get/set those default values.

Note that you can override these default values and/or pass additional environment variables for each call of cgi script. The following environment variables are set by default.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER_SOFTWARE</td>
<td>cgitest/1.0</td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>localhost</td>
</tr>
<tr>
<td>GATEWAY_INTERFACE</td>
<td>CGI/1.1</td>
</tr>
<tr>
<td>SERVER_PROTOCOL</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>SERVER_PORT</td>
<td>80</td>
</tr>
<tr>
<td>REQUEST_METHOD</td>
<td>GET</td>
</tr>
<tr>
<td>REMOTE_HOST</td>
<td>remote</td>
</tr>
<tr>
<td>REMOTE_ADDR</td>
<td>127.0.0.1</td>
</tr>
</tbody>
</table>

**call-with-cgi-script** (script proc :key (environment () (parameters #f)) (parameters #f))

{www.cgi.test} Runs a script with given environment, and calls proc with one argument, an input port which is connected to the pipe of script’s standard output. The argument script should be a list of program name and its arguments. Each element are passed to x->string first to stringify. The script is run under the environment given by environment variable and the default test environment described above. The environment argument must be an associative list, in which each key (car) is the name of the environment variable and its cdr is the value. Both are passed to x->string first. If the same environment variable appears in environment and the default test environment, the one in environment is used. Additionally, if an associative list is given to the parameters argument, a query string is built from it and passed the script. The actual method to pass the query string depends on the value of REQUEST_METHOD environment variable in the setting. If REQUEST_METHOD is either GET or HEAD, the query string is put in an environment variable QUERY_STRING. If it is POST, the query string is fed to the standard input of the script. In the latter case, CONTENT_TYPE is set to application/x-www-form-urlencoded and CONTENT_LENGTH are set to the length of QUERY_STRING automatically. If REQUEST_METHOD is other values, parameters is ignored. You can bypass this mechanism and set up environment variable QUERY_STRING directly, if you wish.

**run-cgi-script->header&body** script reader :key environment parameters

{www.cgi.test} A convenient wrapper of call-with-cgi-script. The script, environment and parameters are passed to call-with-cgi-script as they are. The output of the script is parsed by run-cgi-script->header&body. First, the RFC2822 header fields are parsed by rfc822-read-headers (see Section 12.41 [RFC822 message parsing], page 917). Then, the
**reader** is called with an input port which is piped to the script’s output. **Run-cgi-script->header&body** returns two values, the list of headers (as parsed by rfc822-read-headers), and the return value of **reader**.

**run-cgi-script->sxml**  
```scheme
script :key environment parameters
{www.cgi.test} This is a procedure that uses ssax:xml->sxml (see Section 12.59 [Functional XML parser], page 959) as the reader in run-cgi-script->header&body. Useful when you’re testing a cgi script that produces well-formed HTML and/or XML document.
```

**run-cgi-script->string**  
**run-cgi-script->string-list**  
```scheme
{www.cgi.test} These procedures use port->string and port->string-list (see Section 6.21.7.4 [Input utility functions], page 274) as the reader in run-cgi-script->header&body, respectively.
```

An example:

```scheme
(run-cgi-script->string-list "bbs.cgi"
 :environment '((REMOTE_ADDR . "12.34.56.78")
 :parameters '((command . "view")
 (page . 1234)))
```

### 12.96 **www.css** - CSS parsing and construction

**www.css**  
This module provides tools to convert between S-expression and CSS.

The S-expression CSS (SxCSS) is a convenient way to manipulate CSS in Scheme.

For example, the following CSS and SxCSS are equivalent, and can be converted back and forth:

**CSS:**

```css
body { padding-left: 11em;  
    font-family: Georgia, "Times New Roman", Times, serif;  
    color: purple;  
    background-color: #d8da3d }
ul.navbar li { background: white;  
    margin: 0.5em 0;  
    padding: 0.3em;  
    border-right: 1em solid black }
ul#spec > a { text-decoration: none }
a:visited { color: purple !important }
```

**SxCSS:**

```scheme
((style-rule body  
    (padding-left (11 em))
    (font-family (:or Georgia "Times New Roman" Times serif))
    (color purple)
    (background-color (color "d8da3d")))
(style-rule ((ul (class navbar)) li)
    (background white)
    (margin #((0.5 em) 0))
    (padding (0.3 em))
    (border-right #((1 em) solid black)))
(style-rule ((ul (id spec)) > a) (text-decoration none))
```
(style-rule (a (: visited)) (color purple !important))

See the “CSS in S-expressions” section below for the complete specification.

Constructing CSS

construct-css sxcss :optional oport
{www.css} Take SxCSS and writes out CSS to the given port, defaulted to the current output port.

Parsing CSS

parse-css :optional iport
{www.css} Read CSS from the given port, defaulted to the current input port, and returns SxCSS.

When it encounters unparsable CSS (either a malformed CSS, or unsupported syntax), it emits a warning message, ignore the unparsable part and tries to continue reading the rest.

NB: Currently we don’t handle @charset directive; we assume the text is already in the port’s encoding. We may support it in future versions.

parse-css-file file :key encoding
{www.css} Read the CSS text from the given file and parse it using parse-css. Again, we don’t handle @charset directive yet, and you have to pass encoding argument if the CSS text isn’t in the Gauche’s native character encoding.

parse-css-selector-string str
{www.css} This parses the selector part of the CSS.

(parse-css-selector-string "ul li.item span#foo")
⇒ (ul (li (class item)) (span (id foo)))

(parse-css-selector-string "h1,h2")
⇒ (:or h1 h2)

CSS in S-expression

The following is the complete rules of SxCSS syntax.

<sxcss> : ({<style-rule> | <at-rule>} ...)

<style-rule> : (style-rule <pattern> <declaration> ...) | (style-decls <declaration> ...)

<pattern> : <selector> | (:or <selector> ...)

<selector> : <simple-selector> | <chained-selector>

<chained-selector> : (<simple-selector> . (<op>? . <chained-selector>))

<op> : > | + | ~

<simple-selector> : <element-name> | (<element-name> <option> ...)

<option> : (id <name>) ; E#id

| (class <ident>) ; E.class
| (has <ident>) ; E[attrib]
| (= <ident> <attrib-value>) ; E[attrib=val]
| (|= <ident> <attrib-value>) ; E[attrib="val]
| (== <ident> <attrib-value>) ; E[attrib|=val]
| ("= <ident> <attrib-value>) ; E[attrib=val]
| ($) <ident> <attrib-value>) ; E[attrib$=val]
| (:not <negation-arg>) ; E:not(s)
| (: <ident>) ; E:pseudo-class
Chapter 12: Library modules - Utilities

---

| (<fn> <ident> ...) | E::pseudo-class(arg) |
| (:: <ident>) | E::pseudo-element |

<element-name> : <ident> | * |
<attrib-value> : <ident> | <string> |
<negation-arg> | <element-name> | * | <option> ; except <negation-arg> |

<declaration> : (<ident> <expr> <expr2> ... <important>?) |
!important : !important |
<expr> : <term> |
| (/ <term> <term> ...)
| (:or <term> <term> ...)
| #(<term> <term> ...); juxtaposition |
<term> : <quantity> | (- <quantity>) | (+ <quantity>) |
| <string> | <ident> | <url> | <hexcolor> | <function> |
<quantity> : <number> |
| (#(<number> %))
| ((<number> <ident>)) |
<url> | (url <string>) |
<hexcolor> | (color <string>) ; <string> must be hexdigits |
<function> | (<fn> <arg> ...)
<arg> | <term> | #(<term> ...) | (/ <term> <term> ...)

<at-rule> : <at-media-rule> | <at-import-rule> |
| ; NB: Other at-rules are not supported yet |
<at-media-rule> : (@media (<symbol> ...) <style-rule> ...)
<at-import-rule> : (@import <string> (<symbol> ...))

NB: Negation op is :not instead of not, since (not <negation-arg>) would be ambiguous from the simple node named "not" with one option.

NB: style-decls selector rule is currently won’t appear in the parse-cs output; it can be used in SxCSS to make construct-cs render declarations only, which can be used in the style attribute of the document, for example.

(with-output-to-string
  (cut construct-cs
    '((style-decls (width (50 %))
        (padding #(0 (10 pt) 0 (10 pt)))))
  )
⇒ "width:50%;padding:0 10pt 0 10pt"
Appendix A  C to Scheme mapping

For the convenience of the programmers familiar to C, I composed a simple table of C operators and library functions with the corresponding Scheme functions.

+  R7RS arithmetic procedure +. See Section 6.3.4 [Arithmetics], page 126.

*  R7RS arithmetic procedure *. See Section 6.3.4 [Arithmetics], page 126.

+=  R7RS arithmetic procedure +=. See Section 6.3.4 [Arithmetics], page 126.

-=  R7RS arithmetic procedure -=. See Section 6.3.4 [Arithmetics], page 126.

->  R7RS arithmetic procedure ->. See Section 6.3.4 [Arithmetics], page 126.

*  (binary)  R7RS arithmetic procedure *. See Section 6.3.4 [Arithmetics], page 126.

*  (unary) No equivalent procedure. Scheme doesn’t have explicit notation of pointers.

*  No equivalent procedure.

/  R7RS arithmetic procedure / and /=. See Section 6.3.4 [Arithmetics], page 126. Unlike C operator, Scheme version is transitive.

/=  R7RS arithmetic procedure /=. See Section 6.3.4 [Arithmetics], page 126. Unlike C operator, Scheme version is transitive.

& (binary)  Gauche logand. See Section 10.3.22 [R7RS bitwise operations], page 664.

& (unary) No equivalent procedure. Scheme doesn’t have explicit notation of pointers.

&&  R7RS syntax and. See Section 4.5 [Conditionals], page 55.

&=  R7RS syntax set!. See Section 4.4 [Assignments], page 52.

|  Gauche logior. See Section 10.3.22 [R7RS bitwise operations], page 664.

||  R7RS syntax or. See Section 4.5 [Conditionals], page 55.

|=  No equivalent procedure.

^  Gauche logxor. See Section 10.3.22 [R7RS bitwise operations], page 664.

=  R7RS syntax set!. See Section 4.4 [Assignments], page 52.

==  R7RS equivalence procedure, eq?, eqv? and equal?. See Section 6.2.1 [Equality], page 110.

<  R7RS arithmetic procedure < and <=. See Section 6.3.3 [Numerical comparison], page 125. Unlike C operator, Scheme version is transitive.

<=  Gauche ash. See Section 10.3.22 [R7RS bitwise operations], page 664.

<<  No equivalent procedure.

>  R7RS arithmetic procedure > and >>. See Section 6.3.3 [Numerical comparison], page 125. Unlike C operator, Scheme version is transitive.

>>  Gauche ash. See Section 10.3.22 [R7RS bitwise operations], page 664.

>>=  No equivalent procedure.
Appendix A: C to Scheme mapping

%  R7RS operator modulo and remainder. See Section 6.3.4 [Arithmetics], page 126.
%=

[]  R7RS vector-ref (see Section 6.13.1 [Vectors], page 195) is something close. Or you can use Gauche's generic function ref (see Section 9.31 [Sequence framework], page 510) for arbitrary sequences.

.  Gauche slot-ref is something close to this. See Section 7.3.2 [Accessing instance], page 346.

~  Gauche lognot. See Section 10.3.22 [R7RS bitwise operations], page 664.

~=

!  R7RS procedure not. See Section 6.4 [Booleans], page 139.

!=

abort  Gauche sys-abort. See Section 6.24.1 [Program termination], page 293.

abs  R7RS abs. See Section 6.3.4 [Arithmetics], page 126.

access  Gauche sys-access. See Section 6.24.4.4 [File stats], page 301.

acos  R7RS acos. See Section 6.3.4 [Arithmetics], page 126.

alarm  Gauche sys-alarm. See Section 6.24.14 [Miscellaneous system calls], page 324.

asctime  Gauche sys-asctime. See Section 6.24.9 [Time], page 316.

asin  R7RS asin. See Section 6.3.4 [Arithmetics], page 126.

assert  No equivalent function in Gauche.

atan  R7RS atan. See Section 6.3.4 [Arithmetics], page 126.

atan2

atexit  No equivalent function in Gauche, but the "after" thunk of active dynamic handlers are called when exit is called. See Section 6.24.1 [Program termination], page 293, and See Section 6.15.7 [Continuations], page 226.

atof

atoi

 atol  You can use string->number. See Section 6.3.5 [Numerical conversions], page 134.

bsearch  You can use SRFI-133's vector-binary-search. See Section 10.3.2 [R7RS vectors], page 597.

calloc  Allocation is handled automatically in Scheme.

ceil  R7RS ceiling. See Section 6.3.4 [Arithmetics], page 126.

cfgetispeed

cfgetospeed

cfsetispeed


chdir  Gauche's sys-chdir. See Section 6.24.4.5 [Other file operations], page 304.

chmod  Gauche's sys-chmod. See Section 6.24.4.4 [File stats], page 301.
**clearerr**  Not supported yet.

**clock**  No equivalent function in Gauche. You can use `sys-times` to get information about CPU time.

**close**  You can’t directly close the file descriptor, but when you use `close-input-port` or `close-output-port`, underlying file is closed. Some port-related functions, such as `call-with-output-file`, automatically closes the file when operation is finished. The file is also closed when its governing port is garbage collected. See Section 6.21.3 [Common port operations], page 260.

**closedir**  No equivalent function in Gauche. You can use `sys-readdir` to read the directory entries at once. See Section 6.24.4.1 [Directories], page 296.

**cos**  See Section 6.3.4 [Arithmetics], page 126.

**cosh, cosh**  cos and cosh. See Section 6.3.4 [Arithmetics], page 126.

**creat**  A file is implicitly created by default when you open it for writing. See Section 6.21.4 [File ports], page 264, for more control over the creation of files.

**ctermid**  Gauche `sys-ctermid`. See Section 6.24.8 [System inquiry], page 313.

**ctime**  Gauche `sys-ctime`. See Section 6.24.9 [Time], page 316.

**cuserid**  No equivalent function. This is removed from the newer POSIX. You can use alternative functions, such as `sys-getlogin` or `sys-getpwuid` with `sys-getuid`.

**difftime**  Gauche `sys-difftime`. See Section 6.24.9 [Time], page 316.

**div**  You can use R7RS `quotient` and `remainder`. See Section 6.3.4 [Arithmetics], page 126.

**dup**  Not directly supported, but you can use `port-fd-dup`.

**dup2**  Not directly supported, but you can use `port-fd-dup`.

**execcl, execle, execclp, execv, execve, execvp**  Gauche `sys-exec`. See Section 6.24.10 [Process management], page 318. For higher level interface, Section 9.27 [High-level process interface], page 488.

**exit**  Use `exit` or `sys-exit`, depends on what you need. See Section 6.24.1 [Program termination], page 293.

**exp**  R7RS `exp`. See Section 6.3.4 [Arithmetics], page 126.

**fabs**  R7RS `abs`. See Section 6.3.4 [Arithmetics], page 126.

**fclose**  You can’t directly close the file stream, but when you use `close-input-port` or `close-output-port`, underlying file is closed. Some port-related functions, such as `call-with-output-file`, automatically closes the file when operation is finished. The file is also closed when its governing port is garbage collected.

**fcntl**  Implemented as `sys-fcntl` in `gauche.fcntl` module. See Section 9.11 [Low-level file operations], page 432.

**fdopen**  Gauche’s `open-input-fd-port` or `open-output-fd-port`. See Section 6.21.4 [File ports], page 264.
feof  No equivalent operation, but you can check if an input port have reached to the end by `peek-char` or `peek-byte`. See Section 6.21.7.1 [Reading data], page 270.
ferror Not supported yet.
fflush Gauche’s `flush`. See Section 6.21.8 [Output], page 275.
fgets Use `read-char` or `read-byte`. See Section 6.21.7 [Input], page 270.
fgets Use Gauche’s `port-tell` (see Section 6.21.3 [Common port operations], page 260)
fgets Use `read-line` or `read-string`. See Section 6.21.7 [Input], page 270.
fileno `port-file-number`. See Section 6.21.3 [Common port operations], page 260.
floor R7RS `floor`. See Section 6.3.4 [Arithmetics], page 126.
mod Gauche’s `fmod`.
open R7RS `open-input-file` or `open-output-file` corresponds to this operation. See Section 6.21.4 [File ports], page 264.
forkpty Use `sys-forkpty`. See Section 9.33 [Terminal control], page 518.
fcprintf Not supported.
fprintf Not directly supported, but Gauche’s `format` provides similar functionality. See Section 6.21.8 [Output], page 275. SLIB has `printf` implementation.
ftputc Use `write-char` or `write-byte`. See Section 6.21.8 [Output], page 275.
ftpuses Use `display`. See Section 6.21.8 [Output], page 275.
fwrite Not directly supported. To read binary numbers, see Section 12.1 [Binary I/O], page 808. If you want to read a chunk of bytes, you may be able to use `read-uvector!`. See Section 9.38.4 [Uvector block I/O], page 567.
free You don’t need this in Scheme.
freopen Not supported.
frexp Gauche’s `frexp`.
fscanf Not supported. For general case, you have to write a parser. If you can keep the data in S-exp, you can use `read`. If the syntax is very simple, you may be able to utilize `string-tokenize` in `srfi.14` (Section 11.6 [String library], page 692), and/or regular expression stuff (Section 6.12 [Regular expressions], page 183).
fsync Use Gauche’s `port-seek` (see Section 6.21.3 [Common port operations], page 260)
fssetpos Use Gauche’s `port-seek` (see Section 6.21.3 [Common port operations], page 260)
fsfstat Gauche’s `sys-stat`. See Section 6.24.4.4 [File stats], page 301.
ftell Use Gauche’s `port-tell` (see Section 6.21.3 [Common port operations], page 260)
fwrite Not directly supported. To write binary numbers, see Section 12.1 [Binary I/O], page 808. If you want to write a chunk of bytes, you may simply use `display` or `write-uvector` (see Section 9.38.4 [Uvector block I/O], page 567).
getc Use `read-char` or `read-byte`. See Section 6.21.7 [Input], page 270.
getcwd Gauche’s `sys-getcwd`. See Section 6.24.8 [System inquiry], page 313.
getdomainname
Gauche’s sys-getdomainname. See Section 6.24.8 [System inquiry], page 313.

getegid
Gauche’s sys-getegid. See Section 6.24.8 [System inquiry], page 313.

getenv
Gauche’s sys-getenv. See Section 6.24.3 [Environment inquiry], page 295.

geteuid
Gauche’s sys-geteuid. See Section 6.24.8 [System inquiry], page 313.

gethostname
Gauche’s sys-gethostname. See Section 6.24.8 [System inquiry], page 313.

getgid
Gauche’s sys-getgid. See Section 6.24.8 [System inquiry], page 313.

getgrgid
Gauche’s sys-getgrgid and sys-getgrnam. See Section 6.24.5 [Unix groups and users], page 304.

groups
Gauche’s sys-getgroups. See Section 6.24.8 [System inquiry], page 313.

getlogin
Gauche’s sys-getlogin. See Section 6.24.8 [System inquiry], page 313.

getgpgp
Gauche’s sys-getgpgp. See Section 6.24.8 [System inquiry], page 313.

getpgrp
Gauche’s sys-getpgrp. See Section 6.24.8 [System inquiry], page 313.

getppid
Gauche’s sys-getppid. See Section 6.24.8 [System inquiry], page 313.

getpwnam
Gauche’s sys-getpwnam and sys-getpwuid. See Section 6.24.5 [Unix groups and users], page 304.

gets
Use read-line or read-string. See Section 6.21.7 [Input], page 270.

gettimeofday
Gauche’s sys-gettimeofday. See Section 6.24.9 [Time], page 316.

gteuid
Gauche’s sys-geteuid. See Section 6.24.8 [System inquiry], page 313.

gmtime
Gauche’s sys-gmtime. See Section 6.24.9 [Time], page 316.

isalnum
Not directly supported, but you can use R7RS char-alphabetic? and char-numeric?. See Section 6.9 [Characters], page 160. You can also use character set. See Section 6.10 [Character sets], page 165, also Section 10.3.6 [R7RS character sets], page 614.

isalpha
R7RS char-alphabetic?. See Section 6.9 [Characters], page 160. See also Section 6.10 [Character sets], page 165, and Section 10.3.6 [R7RS character sets], page 614.

isatty
Gauche’s sys-isatty. See Section 6.24.4.5 [Other file operations], page 304.

iscntrl
Not directly supported, but you can use (char-set-contains? char-set:iso-control c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

isdigit
R7RS char-numeric?. See Section 6.9 [Characters], page 160. You can also use (char-set-contains? char-set:digit c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

isgraph
Not directly supported, but you can use (char-set-contains? char-set:graphic c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

islower
R7RS char-lower-case?. See Section 6.9 [Characters], page 160. You can also use (char-set-contains? char-set:lower-case c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.
isprint  Not directly supported, but you can use (char-set-contains? char-set:printing c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

ispunct  Not directly supported, but you can use (char-set-contains? char-set:punctuation c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

isspace  R7RS char-whitespace?. See Section 6.9 [Characters], page 160. You can also use (char-set-contains? char-set:whitespace c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

isupper  R7RS char-upper-case?. See Section 6.9 [Characters], page 160. You can also use (char-set-contains? char-set:upper-case c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

isxdigit  Not directly supported, but you can use (char-set-contains? char-set:hex-digit c) with srfi.14. See Section 10.3.6 [R7RS character sets], page 614.

kill  Gauche's sys-kill. See Section 6.24.7 [Signal], page 306.

labs  R7RS abs. See Section 6.3.4 [Arithmetics], page 126.

ldexp  Gauche's ldexp.

ldiv  Use R7RS quotient and remainder. See Section 6.3.4 [Arithmetics], page 126.

link  Gauche's sys-link. See Section 6.24.4.2 [Directory manipulation], page 299.

localeconv  Gauche's sys-localeconv. See Section 6.24.6 [Locale], page 306.

localtime  Gauche's sys-localtime. See Section 6.24.9 [Time], page 316.

log  R7RS log. See Section 6.3.4 [Arithmetics], page 126.

log10  Not directly supported. log10(z) = (/ (log z) (log 10)).

longjmp  R7RS call/cc provides similar (superior) mechanism. See Section 6.15.7 [Continuations], page 226.

lseek  Use Gauche’s port-seek (see Section 6.21.3 [Common port operations], page 260)

malloc  Not necessary in Scheme.

mblen  mbstowcs

mbtowc  Gauche handles multibyte strings internally, so generally you don’t need to care about multibyte-ness of the string. string-length always returns a number of characters for a string in supported encoding. If you want to convert the character encoding, see Section 9.5 [Character code conversion], page 399.

memcmp  memcpy  memmove  memset  No equivalent functions.

mkdir  Gauche’s sys-mkdir. See Section 6.24.4.2 [Directory manipulation], page 299.

mkfifo  Gauche’s sys-mkfifo.

mkstemp  Gauche’s sys-mkstemp. See Section 6.24.4.2 [Directory manipulation], page 299. Use this instead of tmpnam.
mktime  Gauche’s sys-mktime. See Section 6.24.9 [Time], page 316.

modf  Gauche’s modf.

open  Not directly supported. R7RS open-input-file or open-output-file corresponds to this operation. See Section 6.21.4 [File ports], page 264.

opendir  Not directly supported. You can use sys-readdir to read the directory entries at once. See Section 6.24.4.1 [Directories], page 296.

openpty  Use sys-openpty. See Section 9.33 [Terminal control], page 518.

pathconf  Not supported.

pause  Gauche’s sys-pause. See Section 6.24.14 [Miscellaneous system calls], page 324.

perror  No equivalent function in Gauche. System calls generally throws an error (<system-error>), including the description of the reason of failure.

pipe  Gauche’s sys-pipe. See Section 6.24.4.5 [Other file operations], page 304.

pow  R7RS expt. See Section 6.3.4 [Arithmetics], page 126.

printf  Not directly supported, but Gauche’s format provides similar functionality. See Section 6.21.8 [Output], page 275. SLIB has printf implementation.

putc  Use write-char or write-byte. See Section 6.21.8 [Output], page 275.

puts  Use display. See Section 6.21.8 [Output], page 275.

gsort  Gauche’s sort and sort! provides a convenient way to sort list of items. See Section 6.23 [Sorting and merging], page 291.

raise  No equivalent function in Gauche. Scheme function raise (SRFI-18) is to raise an exception. You can use (sys-kill (sys-getpid) SIG) to send a signal SIG to the current process.

rand  Not supported directly, but on most platforms a better RNG is available as sys-random. See Section 6.24.14 [Miscellaneous system calls], page 324.

read  Not supported directly, but you may be able to use read-uvector or read-uvector! (see Section 9.38.4 [Uvector block I/O], page 567).

readdir  Not supported directly. Gauche’s sys-readdir reads the directory at once. See Section 6.24.4.1 [Directories], page 296.

readlink  Gauche’s sys-readlink. See Section 6.24.4.2 [Directory manipulation], page 299. This function is available on systems that support symbolic links.

realloc  Not necessary in Scheme.

realpath  Gauche’s sys-normalize-pathname or sys-realpath. See Section 6.24.4.3 [Pathnames], page 300.

remove  Gauche’s sys-remove. See Section 6.24.4.2 [Directory manipulation], page 299.

rename  Gauche’s sys-rename. See Section 6.24.4.2 [Directory manipulation], page 299.

rewind  Not directly supported, but you can use port-seek instead. See Section 6.21.3 [Common port operations], page 260.

rewinddir  Not supported directly. You can use sys-readdir to read the directory entries at once. See Section 6.24.4.1 [Directories], page 296.
Appendix A: C to Scheme mapping

rmdir
Gauche's sys-rmdir. See Section 6.24.2 [Directory manipulation], page 299.

scanf
Not supported. For general case, you have to write a parser. If you can keep the data in S-exp, you can use read. If the syntax is very simple, you may be able to utilize string-tokenize in srfi.14 (Section 11.6 [String library], page 692), and/or regular expression stuff (Section 6.12 [Regular expressions], page 183).

select
Gauche's sys-select. See Section 6.24.11 [I/O multiplexing], page 321.

setbuf
Not necessary.

setgid
Gauche's sys-setgid.

setjmp
R7RS call/cc provides similar (superior) mechanism. See Section 6.15.7 [Continuations], page 226.

setlocale
Gauche’s sys-setlocale. See Section 6.24.6 [Locale], page 306.

setpgid
Gauche’s sys-setpgid. See Section 6.24.8 [System inquiry], page 313.

setsid
Gauche’s sys-setsid. See Section 6.24.8 [System inquiry], page 313.

setvbuf
Not necessary.

sigaction
You can use set-signal-handler! to install signal handlers. See Section 6.24.7.3 [Handling signals], page 309.

sigaddset

sigdelset

sigemptyset

sigfillset
Gauche’s sys-sigset-add! and sys-sigset-delete!. See Section 6.24.7.1 [Signals and signal sets], page 307.

sigismember
Not supported yet.

siglongjmp
R7RS call/cc provides similar (superior) mechanism. See Section 6.15.7 [Continuations], page 226.

signal
You can use with-signal-handlers to install signal handlers. See Section 6.24.7.3 [Handling signals], page 309.

sigpending
Not supported yet.

sigprocmask
Signal mask is handled internally. See Section 6.24.7.3 [Handling signals], page 309.

sigsetjmp
R7RS call/cc provides similar (superior) mechanism. See Section 6.15.7 [Continuations], page 226.

sigsuspend
Gauche’s sys-sigsuspend. See Section 6.24.7.4 [Masking and waiting signals], page 311.

sigwait
Gauche’s sys-sigwait. See Section 6.24.7.4 [Masking and waiting signals], page 311.
Appendix A: C to Scheme mapping

sin
 sinh Use sin and sinh. See Section 6.3.4 [Arithmetics], page 126.
sleep Gauche’s sys-sleep. See Section 6.24.14 [Miscellaneous system calls], page 324.
sprintf Not directly supported, but Gauche’s format provides similar functionality. See Section 6.21.8 [Output], page 275. SLIB has printf implementation.
sqrt R7RS sqrt. See Section 6.3.4 [Arithmetics], page 126.
srand Not supported directly, but on most platforms a better RNG is available as sys-random (see Section 6.24.14 [Miscellaneous system calls], page 324). The math.mt-random module provides much superior RNG (see Section 12.36 [Mersenne-Twister random number generator], page 893).
sscanf Not supported. For general case, you have to write a parser. If you can keep the data in S-exp, you can use read. If the syntax is very simple, you may be able to utilize string-tokenize in srfi.14 (Section 11.6 [String library], page 692), and/or regular expression stuff (Section 6.12 [Regular expressions], page 183).
stat Gauche’s sys-stat. See Section 6.24.4.4 [File stats], page 301.
strcsemp R7RS string-ci=? and other comparison functions. See Section 6.11.8 [String comparison], page 177.
strcat R7RS string-append. See Section 6.11.9 [String utilities], page 178.
strchr SRFI-13 string-index. See Section 11.6.7 [SRFI-13 String searching], page 697.
strcmp R7RS string=? and other comparison functions. See Section 6.11.8 [String comparison], page 177.
strcoll Not supported yet.
strncpy SRFI-13 substring. See Section 6.11.9 [String utilities], page 178.
strcspn Not directly supported, but you can use SRFI-13 string-skip with a character set. See Section 11.6.7 [SRFI-13 String searching], page 697.
strftime Gauche’s sys-strftime. See Section 6.24.9 [Time], page 316.
strlen R7RS string-length. See Section 6.11.7 [String accessors & modifiers], page 177.
strncat Not directly supported, but you can use string-append and substring.
strncsemp SRFI-13 string-compare-ci provides the most flexible (but a bit difficult to use) functionality. See Section 11.6.5 [SRFI-13 String comparison], page 695. If what you want is just to check the fixed-length prefixes of two string matches, you can use SRFI-13 string-prefix-ci?.
strncmp SRFI-13 string-compare provides the most flexible (but a bit difficult to use) functionality. See Section 11.6.5 [SRFI-13 String comparison], page 695. If what you want is just to check the fixed-length prefixes of two string matches, you can use SRFI-13 string-prefix?. See Section 11.6.6 [SRFI-13 String prefixes & suffixes], page 696.
strncpy SRFI-13 substring. See Section 6.11.9 [String utilities], page 178.
strpbrk Not directly supported, but you can use SRFI-13 string-skip with a character set. See Section 11.6.7 [SRFI-13 String searching], page 697.
strchr  SRFI-13 string-index-right. See Section 11.6.7 [SRFI-13 String searching], page 697.

strspn  Not directly supported, but you can use SRFI-13 string-index with a character set. See Section 11.6.7 [SRFI-13 String searching], page 697.

strstr  SRFI-13 string-contains. See Section 11.6.7 [SRFI-13 String searching], page 697.

strtok  You can use R7RS string->number. See Section 6.3.5 [Numerical conversions], page 134.

strtol  SRFI-13 string-tokenize. See Section 11.6.12 [SRFI-13 Other string operations], page 700.

strtol  You can use R7RS string->number. See Section 6.3.5 [Numerical conversions], page 134.

strxfrm  Not supported yet.

symlink  Gauche’s sys-symlink. See Section 6.24.4.2 [Directory manipulation], page 299. This function is available on systems that support symbolic links.

sysconf  Not supported yet.

system  Gauche’s sys-system. See Section 6.24.10 [Process management], page 318. It is generally recommended to use the process library (Section 9.27 [High-level process interface], page 488).

tan  R7RS tan and Gauche tanh. See Section 6.3.4 [Arithmetics], page 126.

tcdrain  Corresponding functions are: sys-tcdrain, sys-tcflow, sys-tcflush, sys-tcgetattr, sys-tcgetpgrp, sys-tcsendbreak, sys-tcsetattr, sys-tcsetpgrp. See Section 9.33 [Terminal control], page 518.

time  Gauche’s sys-time. See Section 6.24.9 [Time], page 316.
	imes  Gauche’s sys-times. See Section 6.24.8 [System inquiry], page 313.
tmpfile  Not exactly supported. See sys-mktemp instead. See Section 6.24.4.2 [Directory manipulation], page 299.
tmpnam  Gauche’s sys-tmpnam. This function is provided since it is in POSIX, but its use is discouraged for the potential security risk. Use sys-mktemp instead. See Section 6.24.4.2 [Directory manipulation], page 299.
tolower  R7RS char-upcase and char-downcase. See Section 6.9 [Characters], page 160.
toupper  Gauche’s sys-ttynum. See Section 6.24.4.5 [Other file operations], page 304.
tzset  Not supported yet.
Appendix A: C to Scheme mapping

<table>
<thead>
<tr>
<th>Function</th>
<th>Scheme Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>umask</td>
<td>Gauche’s <code>sys-umask</code></td>
<td>See Section 6.24.4.2 [Directory manipulation], page 299.</td>
</tr>
<tr>
<td>uname</td>
<td>Gauche’s <code>sys-uname</code></td>
<td>See Section 6.24.8 [System inquiry], page 313.</td>
</tr>
<tr>
<td>ungetc</td>
<td>Not directly supported. You can use <code>peek-char</code> to look one character ahead, instead of pushing back.</td>
<td></td>
</tr>
<tr>
<td>unlink</td>
<td>Gauche’s <code>sys-unlink</code></td>
<td>See Section 6.24.4.2 [Directory manipulation], page 299.</td>
</tr>
<tr>
<td>utime</td>
<td>Gauche’s <code>sys-utime</code></td>
<td>See Section 6.24.4.4 [File stats], page 301.</td>
</tr>
<tr>
<td>va_arg</td>
<td>Not necessary, for Scheme handles variable number of arguments naturally.</td>
<td></td>
</tr>
<tr>
<td>va_end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>va_start</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vfprintf</td>
<td></td>
<td>Not directly supported, but Gauche’s <code>format</code> provides similar functionality. See Section 6.21.8 [Output], page 275. SLIB has <code>printf</code> implementation.</td>
</tr>
<tr>
<td>vprintf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsprintf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait</td>
<td>Gauche’s <code>sys-wait</code></td>
<td>See Section 6.24.10 [Process management], page 318.</td>
</tr>
<tr>
<td>waitpid</td>
<td>Gauche’s <code>sys-waitpid</code></td>
<td>See Section 6.24.10 [Process management], page 318.</td>
</tr>
<tr>
<td>wcstombs</td>
<td></td>
<td>Gauche handles multibyte strings internally, so generally you don’t need to care about multibyte-ness of the string. <code>string-length</code> always returns a number of characters for a string in supported encoding. If you want to convert the character encoding, see Section 9.5 [Character code conversion], page 399.</td>
</tr>
<tr>
<td>wctomb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>R7RS <code>display</code> (see Section 6.21.8 [Output], page 275). Or <code>write-uvector</code> (see Section 9.38.4 [Uvector block I/O], page 567).</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B  Function and Syntax Index

! .............................................................. 394

$ ............................................................ 394

$ =>rope ..................................................... 50
$ =>string ................................................... 912
$ =>symbol ................................................... 912
$. ............................................................... 910
$any .......................................................... 911
$assert ......................................................... 913
$between ....................................................... 913
$bind .......................................................... 913
$binding ....................................................... 914
$chain-left .................................................. 916
$chain-right ............................................... 916
$char ........................................................... 911
$char-ci ......................................................... 911
$cut ............................................................ 913
$debug ........................................................ 917
$end-by ........................................................ 916
$eos .......................................................... 911
$expect ........................................................ 913
$fail ........................................................... 910
$fold-parsers ................................................. 915
$fold-parsers-right ....................................... 915
$lazy ............................................................ 917
$let ............................................................. 914
$let* ............................................................ 914
$lift ............................................................. 914
$lift* ............................................................ 914
$list ............................................................ 913
$list* ............................................................ 913
$many ......................................................... 915
$many-till .................................................... 916
$many-till_ .................................................. 916
$many_ ........................................................ 915
$many1 ........................................................ 916
$many1_ ........................................................ 916
$match ........................................................ 911
$match1 ....................................................... 911
$match1* ..................................................... 911
$none-of ....................................................... 911
$not ............................................................ 913
$one-of ......................................................... 911
$optional ..................................................... 913
$or .............................................................. 912
$parameterize ................................................. 916
$raise ........................................................ 910
$repeat ......................................................... 916
$repeat_ ........................................................ 916
$return ......................................................... 910
$satisfy ........................................................ 910
$sep-by ......................................................... 916
$sep-end-by ................................................ 916
$seq ........................................................... 913
$seq_ ........................................................... 913
$string .......................................................... 911
$string-ci ...................................................... 911

% ............................................................. 394
% ............................................................. 394
%= ............................................................. 395
%acos ......................................................... 815
%asin ......................................................... 815
%atan ......................................................... 815
%cos ......................................................... 815
%cosh ......................................................... 815
%cospi ....................................................... 815
%exp ......................................................... 815
%exppt ....................................................... 815
%log ......................................................... 815
%macroexpand ............................................... 104
%macroexpand-1 ........................................... 104
%sin ......................................................... 815
%sinh ......................................................... 815
%sinpi ....................................................... 815
%sqrt ......................................................... 815
%tan ......................................................... 815
%tanh ......................................................... 815
%tanpi ....................................................... 815

& ............................................................. 394

( (setter ~) .................................................... 219
 (setter cgi-test-environment-ref) ......................... 1052
 (setter dict-get) .......................................... 428
 (setter object-apply) .................................... 225
 (setter port-buffering) ................................... 261
 (setter random-data-seed) ............................... 841
 (setter ref) .................................................. 209, 220, 347, 510
 (setter subseq) ............................................ 511

* .............................................................. 126, 394
* .............................................................. 127
* .............................................................. 395

+ .............................................................. 126, 394
+ .............................................................. 127
+ .............................................................. 395

- .............................................................. 127, 394
- .............................................................. 127
- .............................................................. 395
- .............................................................. 394
- .............................................................. 394
->char-set .................................................... 615
<table>
<thead>
<tr>
<th>Function/Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>@vector-cumulate</td>
<td>562</td>
</tr>
<tr>
<td>@vector-div</td>
<td>566</td>
</tr>
<tr>
<td>@vector-div!</td>
<td>566</td>
</tr>
<tr>
<td>@vector-dot</td>
<td>566</td>
</tr>
<tr>
<td>@vector-drop</td>
<td>561</td>
</tr>
<tr>
<td>@vector-drop-right</td>
<td>561</td>
</tr>
<tr>
<td>@vector-drop-while</td>
<td>562</td>
</tr>
<tr>
<td>@vector-drop-while-right</td>
<td>562</td>
</tr>
<tr>
<td>@vector-empty</td>
<td>557</td>
</tr>
<tr>
<td>@vector-every</td>
<td>562</td>
</tr>
<tr>
<td>@vector-fill!</td>
<td>558</td>
</tr>
<tr>
<td>@vector-filter</td>
<td>562</td>
</tr>
<tr>
<td>@vector-fold</td>
<td>561</td>
</tr>
<tr>
<td>@vector-fold-right</td>
<td>561</td>
</tr>
<tr>
<td>@vector-for-each</td>
<td>562</td>
</tr>
<tr>
<td>@vector-index</td>
<td>562</td>
</tr>
<tr>
<td>@vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>@vector-ior!</td>
<td>566</td>
</tr>
<tr>
<td>@vector-ior!</td>
<td>566</td>
</tr>
<tr>
<td>@vector-length</td>
<td>558</td>
</tr>
<tr>
<td>@vector-map</td>
<td>562</td>
</tr>
<tr>
<td>@vector-map!</td>
<td>562</td>
</tr>
<tr>
<td>@vector-mul!</td>
<td>565</td>
</tr>
<tr>
<td>@vector-mul!</td>
<td>565</td>
</tr>
<tr>
<td>@vector-multi-copy!</td>
<td>559</td>
</tr>
<tr>
<td>@vector-partition</td>
<td>562</td>
</tr>
<tr>
<td>@vector-range-check</td>
<td>566</td>
</tr>
<tr>
<td>@vector-ref</td>
<td>201</td>
</tr>
<tr>
<td>@vector-remove</td>
<td>562</td>
</tr>
<tr>
<td>@vector-reverse!</td>
<td>559</td>
</tr>
<tr>
<td>@vector-reverse-copy</td>
<td>559</td>
</tr>
<tr>
<td>@vector-reverse-copy!</td>
<td>559</td>
</tr>
<tr>
<td>@vector-segment</td>
<td>561</td>
</tr>
<tr>
<td>@vector-set!</td>
<td>201</td>
</tr>
<tr>
<td>@vector-skip</td>
<td>562</td>
</tr>
<tr>
<td>@vector-skip-right</td>
<td>562</td>
</tr>
<tr>
<td>@vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>@vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>@vector-swap!</td>
<td>558</td>
</tr>
<tr>
<td>@vector-take</td>
<td>561</td>
</tr>
<tr>
<td>@vector-take-right</td>
<td>561</td>
</tr>
<tr>
<td>@vector-take-right!</td>
<td>561</td>
</tr>
<tr>
<td>@vector-take-while</td>
<td>562</td>
</tr>
<tr>
<td>@vector-take-while!</td>
<td>562</td>
</tr>
<tr>
<td>@vector-take-while-right</td>
<td>562</td>
</tr>
<tr>
<td>@vector-unfold</td>
<td>557</td>
</tr>
<tr>
<td>@vector-unfold!</td>
<td>557, 561</td>
</tr>
<tr>
<td>@vector-unfold-right</td>
<td>557</td>
</tr>
<tr>
<td>@vector-unfold-right!</td>
<td>557, 561</td>
</tr>
<tr>
<td>@vector-xor</td>
<td>566</td>
</tr>
<tr>
<td>@vector-xor!</td>
<td>566</td>
</tr>
<tr>
<td>@vector=?</td>
<td>558</td>
</tr>
<tr>
<td>@vector?</td>
<td>200</td>
</tr>
<tr>
<td>........................................................................</td>
<td>219</td>
</tr>
</tbody>
</table>
### Appendix B: Function and Syntax Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply-methods</td>
<td>358</td>
</tr>
<tr>
<td>apply/av</td>
<td>785</td>
</tr>
<tr>
<td>approx?</td>
<td>126</td>
</tr>
<tr>
<td>apropos</td>
<td>448</td>
</tr>
<tr>
<td>aref</td>
<td>394</td>
</tr>
<tr>
<td>args-fold</td>
<td>709</td>
</tr>
<tr>
<td>arguments-drop</td>
<td>806</td>
</tr>
<tr>
<td>arguments-drop-right</td>
<td>806</td>
</tr>
<tr>
<td>arguments-take</td>
<td>806</td>
</tr>
<tr>
<td>arguments-take-right</td>
<td>806</td>
</tr>
<tr>
<td>arithmetic-shift</td>
<td>665</td>
</tr>
<tr>
<td>arity</td>
<td>224</td>
</tr>
<tr>
<td>arity-at-least-value</td>
<td>225</td>
</tr>
<tr>
<td>array</td>
<td>367</td>
</tr>
<tr>
<td>array-&gt;list</td>
<td>370</td>
</tr>
<tr>
<td>array-&gt;vector</td>
<td>370</td>
</tr>
<tr>
<td>array-add-elements</td>
<td>372</td>
</tr>
<tr>
<td>array-add-elements!</td>
<td>372</td>
</tr>
<tr>
<td>array-concatenate</td>
<td>370</td>
</tr>
<tr>
<td>array-copy</td>
<td>367</td>
</tr>
<tr>
<td>array-div-elements</td>
<td>372</td>
</tr>
<tr>
<td>array-div-elements!</td>
<td>372</td>
</tr>
<tr>
<td>array-div-left</td>
<td>372</td>
</tr>
<tr>
<td>array-div-right</td>
<td>372</td>
</tr>
<tr>
<td>array-end</td>
<td>368</td>
</tr>
<tr>
<td>array-expt</td>
<td>372</td>
</tr>
<tr>
<td>array-flip</td>
<td>371</td>
</tr>
<tr>
<td>array-flip!</td>
<td>371</td>
</tr>
<tr>
<td>array-for-each-index</td>
<td>369</td>
</tr>
<tr>
<td>array-inverse</td>
<td>372</td>
</tr>
<tr>
<td>array-length</td>
<td>368</td>
</tr>
<tr>
<td>array-map</td>
<td>370</td>
</tr>
<tr>
<td>array-map!</td>
<td>370</td>
</tr>
<tr>
<td>array-mul</td>
<td>372</td>
</tr>
<tr>
<td>array-mul-elements</td>
<td>372</td>
</tr>
<tr>
<td>array-mul-elements!</td>
<td>372</td>
</tr>
<tr>
<td>array-negate-elements</td>
<td>373</td>
</tr>
<tr>
<td>array-negate-elements!</td>
<td>373</td>
</tr>
<tr>
<td>array-rank</td>
<td>368</td>
</tr>
<tr>
<td>array-reciprocate-elements</td>
<td>373</td>
</tr>
<tr>
<td>array-reciprocate-elements!</td>
<td>373</td>
</tr>
<tr>
<td>array-ref</td>
<td>368</td>
</tr>
<tr>
<td>array-retabulate</td>
<td>370</td>
</tr>
<tr>
<td>array-rotate-90</td>
<td>371</td>
</tr>
<tr>
<td>array-set!</td>
<td>368</td>
</tr>
<tr>
<td>array-shape</td>
<td>368</td>
</tr>
<tr>
<td>array-size</td>
<td>368</td>
</tr>
<tr>
<td>array-start</td>
<td>368</td>
</tr>
<tr>
<td>array-sub-elements</td>
<td>372</td>
</tr>
<tr>
<td>array-sub-elements!</td>
<td>372</td>
</tr>
<tr>
<td>array-transpose</td>
<td>371</td>
</tr>
<tr>
<td>array?</td>
<td>367</td>
</tr>
<tr>
<td>as-black</td>
<td>686</td>
</tr>
<tr>
<td>as-blue</td>
<td>686</td>
</tr>
<tr>
<td>as-bold</td>
<td>686</td>
</tr>
<tr>
<td>as-cyan</td>
<td>686</td>
</tr>
<tr>
<td>as-green</td>
<td>686</td>
</tr>
<tr>
<td>as-magenta</td>
<td>686</td>
</tr>
<tr>
<td>as-nodeset</td>
<td>970</td>
</tr>
<tr>
<td>as-red</td>
<td>686</td>
</tr>
<tr>
<td>as-underline</td>
<td>686</td>
</tr>
<tr>
<td>as-underline</td>
<td>686</td>
</tr>
<tr>
<td>as-white</td>
<td>686</td>
</tr>
<tr>
<td>as-yellow</td>
<td>686</td>
</tr>
<tr>
<td>Function/Operator</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>autoload</td>
<td>289</td>
</tr>
<tr>
<td>bag</td>
<td>606</td>
</tr>
<tr>
<td>bag-&gt;alist</td>
<td>611</td>
</tr>
<tr>
<td>bag-&gt;list</td>
<td>611</td>
</tr>
<tr>
<td>bag-&gt;set</td>
<td>611</td>
</tr>
<tr>
<td>bag-adjoin</td>
<td>607</td>
</tr>
<tr>
<td>bag-adjoin!</td>
<td>608</td>
</tr>
<tr>
<td>bag-any?</td>
<td>609</td>
</tr>
<tr>
<td>bag-contains?</td>
<td>607</td>
</tr>
<tr>
<td>bag-copy</td>
<td>610</td>
</tr>
<tr>
<td>bag-count</td>
<td>609</td>
</tr>
<tr>
<td>bag-decrement!</td>
<td>613</td>
</tr>
<tr>
<td>bag-delete</td>
<td>608</td>
</tr>
<tr>
<td>bag-delete!</td>
<td>608</td>
</tr>
<tr>
<td>bag-delete-all</td>
<td>608</td>
</tr>
<tr>
<td>bag-delete-all!</td>
<td>608</td>
</tr>
<tr>
<td>bag-difference</td>
<td>612</td>
</tr>
<tr>
<td>bag-difference!</td>
<td>612</td>
</tr>
<tr>
<td>bag-disjoint?</td>
<td>607</td>
</tr>
<tr>
<td>bag-element-comparator</td>
<td>607</td>
</tr>
<tr>
<td>bag-element-count</td>
<td>613</td>
</tr>
<tr>
<td>bag-empty?</td>
<td>607</td>
</tr>
<tr>
<td>bag-every?</td>
<td>609</td>
</tr>
<tr>
<td>bag-filter!</td>
<td>610</td>
</tr>
<tr>
<td>bag-find!</td>
<td>609</td>
</tr>
<tr>
<td>bag-fold</td>
<td>610</td>
</tr>
<tr>
<td>bag-fold-unique</td>
<td>613</td>
</tr>
<tr>
<td>bag-for-each</td>
<td>609</td>
</tr>
<tr>
<td>bag-for-each-unique</td>
<td>613</td>
</tr>
<tr>
<td>bag-increment!</td>
<td>613</td>
</tr>
<tr>
<td>bag-intersection!</td>
<td>612</td>
</tr>
<tr>
<td>bag-intersection</td>
<td>612</td>
</tr>
<tr>
<td>bag-map</td>
<td>609</td>
</tr>
<tr>
<td>bag-member</td>
<td>607</td>
</tr>
<tr>
<td>bag-partition</td>
<td>610</td>
</tr>
<tr>
<td>bag-partition!</td>
<td>610</td>
</tr>
<tr>
<td>bag-product</td>
<td>613</td>
</tr>
<tr>
<td>bag-product!</td>
<td>613</td>
</tr>
<tr>
<td>bag-remove</td>
<td>610</td>
</tr>
<tr>
<td>bag-remove!</td>
<td>610</td>
</tr>
<tr>
<td>bag-replace</td>
<td>607</td>
</tr>
<tr>
<td>bag-replace!</td>
<td>608</td>
</tr>
<tr>
<td>bag-search!</td>
<td>608</td>
</tr>
<tr>
<td>bag-size</td>
<td>609</td>
</tr>
<tr>
<td>bag-sum</td>
<td>612</td>
</tr>
<tr>
<td>bag-sum!</td>
<td>612</td>
</tr>
<tr>
<td>bag-unfold</td>
<td>606</td>
</tr>
<tr>
<td>bag-union</td>
<td>612</td>
</tr>
<tr>
<td>bag-union!</td>
<td>612</td>
</tr>
<tr>
<td>bag-unique-size</td>
<td>613</td>
</tr>
<tr>
<td>bag-xor</td>
<td>612</td>
</tr>
<tr>
<td>bag-xor!</td>
<td>612</td>
</tr>
<tr>
<td>bag&lt;=?</td>
<td>611</td>
</tr>
<tr>
<td>bag?&lt;</td>
<td>611</td>
</tr>
<tr>
<td>bag=?</td>
<td>611</td>
</tr>
<tr>
<td>bag&gt;=?</td>
<td>611</td>
</tr>
<tr>
<td>bag&gt;?</td>
<td>611</td>
</tr>
<tr>
<td>balanced-quotient</td>
<td>664</td>
</tr>
<tr>
<td>balanced-remainder</td>
<td>664</td>
</tr>
<tr>
<td>balanced/</td>
<td>664</td>
</tr>
<tr>
<td>barrier-await</td>
<td>543</td>
</tr>
<tr>
<td>barrier-broken?</td>
<td>543</td>
</tr>
<tr>
<td>barrier-reset!</td>
<td>543</td>
</tr>
<tr>
<td>barrier?</td>
<td>543</td>
</tr>
<tr>
<td>base16-decode</td>
<td>923</td>
</tr>
<tr>
<td>base16-decode-string-to</td>
<td>923</td>
</tr>
<tr>
<td>base16-encode</td>
<td>923</td>
</tr>
<tr>
<td>base16-encode-message</td>
<td>923</td>
</tr>
<tr>
<td>base32-decode</td>
<td>923</td>
</tr>
<tr>
<td>base32-decode-string-to</td>
<td>923</td>
</tr>
<tr>
<td>base32-encode</td>
<td>922</td>
</tr>
<tr>
<td>base32-encode-message</td>
<td>922</td>
</tr>
<tr>
<td>base32hex-decode</td>
<td>923</td>
</tr>
<tr>
<td>base32hex-decode-string-to</td>
<td>923</td>
</tr>
<tr>
<td>base32hex-encode</td>
<td>922</td>
</tr>
<tr>
<td>base32hex-encode-message</td>
<td>922</td>
</tr>
<tr>
<td>base64-&gt;bytevector</td>
<td>777</td>
</tr>
<tr>
<td>base64-decode</td>
<td>922</td>
</tr>
<tr>
<td>base64-decode-bytevector</td>
<td>924</td>
</tr>
<tr>
<td>base64-decode-string</td>
<td>924</td>
</tr>
<tr>
<td>base64-decode-string-to</td>
<td>922</td>
</tr>
<tr>
<td>base64-encode</td>
<td>921</td>
</tr>
<tr>
<td>base64-encode-bytevector</td>
<td>924</td>
</tr>
<tr>
<td>base64-encode-message</td>
<td>921</td>
</tr>
<tr>
<td>base64-encode-string</td>
<td>924</td>
</tr>
<tr>
<td>bcrypt-gensalt</td>
<td>828</td>
</tr>
<tr>
<td>bcrypt-hashpw</td>
<td>828</td>
</tr>
<tr>
<td>begin</td>
<td>62, 392, 399, 716</td>
</tr>
<tr>
<td>begin0</td>
<td>63</td>
</tr>
<tr>
<td>bignum?</td>
<td>125</td>
</tr>
<tr>
<td>bimap-left</td>
<td>431</td>
</tr>
<tr>
<td>bimap-left-delete!</td>
<td>431</td>
</tr>
<tr>
<td>bimap-left-exists?</td>
<td>431</td>
</tr>
<tr>
<td>bimap-left-get</td>
<td>431</td>
</tr>
<tr>
<td>bimap-put!</td>
<td>431</td>
</tr>
<tr>
<td>bimap-right</td>
<td>431</td>
</tr>
<tr>
<td>bimap-right-delete!</td>
<td>431</td>
</tr>
<tr>
<td>bimap-right-exists?</td>
<td>431</td>
</tr>
<tr>
<td>bimap-right-get</td>
<td>431</td>
</tr>
<tr>
<td>binary-heap-clear!</td>
<td>832</td>
</tr>
<tr>
<td>binary-heap-copy</td>
<td>832</td>
</tr>
<tr>
<td>binary-heap-delete!</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-empty?</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-find</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-find-max</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-find-min</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-num-entries</td>
<td>832</td>
</tr>
<tr>
<td>binary-heap-pop-max!</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-pop-min!</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-push!</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-remove!</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-swap-max!</td>
<td>833</td>
</tr>
<tr>
<td>binary-heap-swap-min!</td>
<td>833</td>
</tr>
<tr>
<td>binary-port?</td>
<td>587</td>
</tr>
<tr>
<td>bind</td>
<td>786</td>
</tr>
<tr>
<td>bind/box</td>
<td>786</td>
</tr>
<tr>
<td>bind/list</td>
<td>786</td>
</tr>
<tr>
<td>bind/mv</td>
<td>785</td>
</tr>
<tr>
<td>bindtextdomain</td>
<td>1001</td>
</tr>
<tr>
<td>bit-&gt;boolean</td>
<td>204</td>
</tr>
<tr>
<td>bit-&gt;integer</td>
<td>204</td>
</tr>
<tr>
<td>bit-count</td>
<td>665</td>
</tr>
<tr>
<td>bit-field</td>
<td>157</td>
</tr>
<tr>
<td>bit-field-any?</td>
<td>666</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>bit-vector-ior!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-logical-shift</td>
<td>381</td>
</tr>
<tr>
<td>bit-vector-map!/bool</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-map!/int</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-map-&gt;list/bool</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-map-&gt;list/int</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-map/bool</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-map/int</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-nand</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-nand!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-nor</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-nor!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-not</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-not!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-orc1</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-orc1!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-orc2</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-orc2!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-pad</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-pad-right</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-prefix-length</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-prefix?</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-ref/bool</td>
<td>204</td>
</tr>
<tr>
<td>bit-vector-ref/int</td>
<td>204</td>
</tr>
<tr>
<td>bit-vector-reverse!</td>
<td>378</td>
</tr>
<tr>
<td>bit-vector-reverse-copy!</td>
<td>375</td>
</tr>
<tr>
<td>bit-vector-reverse-copy!</td>
<td>375</td>
</tr>
<tr>
<td>bit-vector-segment</td>
<td>375</td>
</tr>
<tr>
<td>bit-vector-set!</td>
<td>205</td>
</tr>
<tr>
<td>bit-vector-suffix-length</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-suffix?</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-swap</td>
<td>378</td>
</tr>
<tr>
<td>bit-vector-take</td>
<td>375</td>
</tr>
<tr>
<td>bit-vector-take-right</td>
<td>375</td>
</tr>
<tr>
<td>bit-vector-trim</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-trim-right</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-unfold</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-unfold-right</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-value-for-each-index</td>
<td>377</td>
</tr>
<tr>
<td>bit-vector-value-map-index-&gt;list</td>
<td>376</td>
</tr>
<tr>
<td>bit-vector-xor</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector-xor!</td>
<td>380</td>
</tr>
<tr>
<td>bit-vector/bool-range</td>
<td>847</td>
</tr>
<tr>
<td>bit-vector/uint-range</td>
<td>847</td>
</tr>
<tr>
<td>bit-vector=?</td>
<td>375</td>
</tr>
<tr>
<td>bit-vector?</td>
<td>204</td>
</tr>
<tr>
<td>bitwise-and</td>
<td>664</td>
</tr>
<tr>
<td>bitwise-andcl</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-andc2</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-eqv</td>
<td>664</td>
</tr>
<tr>
<td>bitwise-fold</td>
<td>667</td>
</tr>
<tr>
<td>bitwise-for-each</td>
<td>667</td>
</tr>
<tr>
<td>bitwise-if</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-ior</td>
<td>664</td>
</tr>
<tr>
<td>bitwise-merge</td>
<td>719</td>
</tr>
<tr>
<td>bitwise-nand</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-nor</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-not</td>
<td>664</td>
</tr>
<tr>
<td>bitwise-orc1</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-orc2</td>
<td>665</td>
</tr>
<tr>
<td>bitwise-unfold</td>
<td>667</td>
</tr>
<tr>
<td>bitwise-xor</td>
<td>664</td>
</tr>
<tr>
<td>blob-&gt;sint-list</td>
<td>725</td>
</tr>
<tr>
<td>blob-&gt;u8-list</td>
<td>725</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td><code>bytevector-uint-ref</code></td>
<td>679</td>
</tr>
<tr>
<td><code>bytevector-uint-set!</code></td>
<td>679</td>
</tr>
<tr>
<td><code>bytevector=?</code></td>
<td>570</td>
</tr>
<tr>
<td><code>bytevector?</code></td>
<td>569</td>
</tr>
<tr>
<td><code>bytrestring-replace</code></td>
<td>778</td>
</tr>
<tr>
<td><code>c128vector-unfold</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c128vector-unfold!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c128vector-unfold-right</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c128vector-unfold-right!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c128vector=</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c128vector?</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32?</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32ite-u8vector</code></td>
<td>568</td>
</tr>
<tr>
<td><code>c32ke-u8vector-generator</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32vector-&gt;list</code></td>
<td>563</td>
</tr>
<tr>
<td><code>c32vector-&gt;vector</code></td>
<td>564</td>
</tr>
<tr>
<td><code>c32vector-add</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-add!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-any</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-append</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-append-subvectors</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-compare</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32vector-concatenate</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c32vector-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c32vector-count</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-cumulate</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-div</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c32vector-div!</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c32vector-dot</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c32vector-drop</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-drop-right</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-drop-right!</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-drop-right!</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-empty?</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32vector-extend</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-filter</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32vector-fold</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-for-each</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-index</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-index-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-length</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32vector-map</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-mul</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-mul!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-multi-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c32vector-partition</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-ref</code></td>
<td>201</td>
</tr>
<tr>
<td><code>c32vector-remove</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-reverse</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c32vector-reverse-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c32vector-reverse-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c32vector-set!</code></td>
<td>201</td>
</tr>
<tr>
<td><code>c32vector-skip</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-skip-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c32vector-sub</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-sub!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c32vector-swap</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32vector-take</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-take-right</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-take-right!</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-take-right!</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c32vector-unfold</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32vector-unfold!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32vector-unfold!</code></td>
<td>557</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td><code>c32vector-unfold-right</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32vector-unfold-right!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c32vector=</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32vector?</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c32vector?</code></td>
<td>200</td>
</tr>
<tr>
<td><code>c64?</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64ite-u8vector</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64ke-u8vector-generator</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-&gt;list</code></td>
<td>563</td>
</tr>
<tr>
<td><code>c64vector-&gt;vector</code></td>
<td>564</td>
</tr>
<tr>
<td><code>c64vector-add</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-add!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-any</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-append</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-append-subvectors</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-compare</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-concatenate</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-count</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-cumulate</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-div</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c64vector-div</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c64vector-dot</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c64vector-drop</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-drop-right</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-drop-while</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-drop-while-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-empty?</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-every</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-fill!</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-filter</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-fold</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-for-each</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-index</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-index-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-length</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-map</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-map!</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-mul</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-mul!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-multi-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-partition</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-ref</code></td>
<td>201</td>
</tr>
<tr>
<td><code>c64vector-remove</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-reverse!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-reverse-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-reverse-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-segment</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-set!</code></td>
<td>201</td>
</tr>
<tr>
<td><code>c64vector-skip</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-skip-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-sub</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-sub!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-swap!</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-take</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-take-right</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-take-right!</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-unfold</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-unfold!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-unfold-right</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-unfold-right!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-&gt;list</code></td>
<td>563</td>
</tr>
<tr>
<td><code>c64vector-&gt;vector</code></td>
<td>564</td>
</tr>
<tr>
<td><code>c64vector-add</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-add!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-any</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-append</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-append-subvectors</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-compare</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-concatenate</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-count</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-cumulate</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-div</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c64vector-div!</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c64vector-dot</code></td>
<td>566</td>
</tr>
<tr>
<td><code>c64vector-drop</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-drop-right</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-drop-while</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-drop-while-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-empty?</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-every</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-fill!</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-filter</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-fold</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-for-each</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-index</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-index-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-length</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-map</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-map!</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-mul</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-mul!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-multi-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-partition</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-ref</code></td>
<td>201</td>
</tr>
<tr>
<td><code>c64vector-remove</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-reverse!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-reverse-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-reverse-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-segment</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-set!</code></td>
<td>201</td>
</tr>
<tr>
<td><code>c64vector-skip</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-skip-right</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-sub</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-sub!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-swap!</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-take</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-take-right</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-take-right!</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-unfold</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-unfold!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-unfold-right</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-unfold-right!</code></td>
<td>557</td>
</tr>
<tr>
<td><code>c64vector-&gt;list</code></td>
<td>563</td>
</tr>
<tr>
<td><code>c64vector-&gt;vector</code></td>
<td>564</td>
</tr>
<tr>
<td><code>c64vector-add</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-add!</code></td>
<td>565</td>
</tr>
<tr>
<td><code>c64vector-any</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-append</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-append-subvectors</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-compare</code></td>
<td>558</td>
</tr>
<tr>
<td><code>c64vector-concatenate</code></td>
<td>561</td>
</tr>
<tr>
<td><code>c64vector-copy</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-copy!</code></td>
<td>559</td>
</tr>
<tr>
<td><code>c64vector-count</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-cumulate</code></td>
<td>562</td>
</tr>
<tr>
<td><code>c64vector-div</code></td>
<td>566</td>
</tr>
<tr>
<td>Function</td>
<td>Page(s)</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>case</td>
<td>56, 392</td>
</tr>
<tr>
<td>case-lambda</td>
<td>51</td>
</tr>
<tr>
<td>case-lambda/tag</td>
<td>803</td>
</tr>
<tr>
<td>case-procedure</td>
<td>806</td>
</tr>
<tr>
<td>case-receive</td>
<td>784</td>
</tr>
<tr>
<td>case/falthrough</td>
<td>392</td>
</tr>
<tr>
<td>cast</td>
<td>395</td>
</tr>
<tr>
<td>cdaaar</td>
<td>144</td>
</tr>
<tr>
<td>cdaadr</td>
<td>144</td>
</tr>
<tr>
<td>cdaar</td>
<td>144</td>
</tr>
<tr>
<td>cdadar</td>
<td>144</td>
</tr>
<tr>
<td>cdadder</td>
<td>144</td>
</tr>
<tr>
<td>cdadr</td>
<td>144</td>
</tr>
<tr>
<td>cdar</td>
<td>144</td>
</tr>
<tr>
<td>cddaar</td>
<td>144</td>
</tr>
<tr>
<td>cddadr</td>
<td>144</td>
</tr>
<tr>
<td>cddaddr</td>
<td>144</td>
</tr>
<tr>
<td>cddar</td>
<td>144</td>
</tr>
<tr>
<td>cdr</td>
<td>143</td>
</tr>
<tr>
<td>ceiling</td>
<td>131</td>
</tr>
<tr>
<td>ceiling-&gt;exact</td>
<td>131</td>
</tr>
<tr>
<td>ceiling-quotient</td>
<td>663</td>
</tr>
<tr>
<td>ceiling-remainder</td>
<td>663</td>
</tr>
<tr>
<td>ces-conversion-supported?</td>
<td>400</td>
</tr>
<tr>
<td>ces-convert</td>
<td>403</td>
</tr>
<tr>
<td>ces-convert-to</td>
<td>403</td>
</tr>
<tr>
<td>ces-equivalent?</td>
<td>400</td>
</tr>
<tr>
<td>ces-guess-from-string</td>
<td>402</td>
</tr>
<tr>
<td>ces-upper-compatible?</td>
<td>400</td>
</tr>
<tr>
<td>cf$</td>
<td>419</td>
</tr>
<tr>
<td>cf-arg-enable</td>
<td>416</td>
</tr>
<tr>
<td>cf-arg-var</td>
<td>419</td>
</tr>
<tr>
<td>cf-arg-with</td>
<td>416</td>
</tr>
<tr>
<td>cf-check-decl</td>
<td>422</td>
</tr>
<tr>
<td>cf-check-decls</td>
<td>422</td>
</tr>
<tr>
<td>cf-check-func</td>
<td>423</td>
</tr>
<tr>
<td>cf-check-funcs</td>
<td>423</td>
</tr>
<tr>
<td>cf-check-header</td>
<td>420</td>
</tr>
<tr>
<td>cf-check-headers</td>
<td>421</td>
</tr>
<tr>
<td>cf-check-lib</td>
<td>423</td>
</tr>
<tr>
<td>cf-check-member</td>
<td>422</td>
</tr>
<tr>
<td>cf-check-members</td>
<td>422</td>
</tr>
<tr>
<td>cf-check-prog</td>
<td>419</td>
</tr>
<tr>
<td>cf-check-type</td>
<td>421</td>
</tr>
<tr>
<td>cf-check-types</td>
<td>421</td>
</tr>
<tr>
<td>cf-config-headers</td>
<td>425</td>
</tr>
<tr>
<td>cf-decl-available?</td>
<td>422</td>
</tr>
<tr>
<td>cf-define</td>
<td>418</td>
</tr>
<tr>
<td>cf-defined?</td>
<td>418</td>
</tr>
<tr>
<td>cf-echo</td>
<td>418</td>
</tr>
<tr>
<td>cf-feature-ref</td>
<td>417</td>
</tr>
<tr>
<td>cf-func-available?</td>
<td>423</td>
</tr>
<tr>
<td>cf-have-subst?</td>
<td>419</td>
</tr>
<tr>
<td>cf-header-available?</td>
<td>420</td>
</tr>
<tr>
<td>cf-help-string</td>
<td>417</td>
</tr>
<tr>
<td>cf-includes-default</td>
<td>421</td>
</tr>
<tr>
<td>cf-init</td>
<td>415</td>
</tr>
<tr>
<td>cf-init-gauche-extension</td>
<td>415</td>
</tr>
<tr>
<td>cf-lang</td>
<td>424</td>
</tr>
<tr>
<td>cf-lang-call</td>
<td>424</td>
</tr>
<tr>
<td>cf-lang-io-program</td>
<td>424</td>
</tr>
<tr>
<td>cf-lang-program</td>
<td>424</td>
</tr>
<tr>
<td>cf-lang-program</td>
<td>424</td>
</tr>
<tr>
<td>cf-lib-available?</td>
<td>423</td>
</tr>
<tr>
<td>cf-make-gpd</td>
<td>426</td>
</tr>
<tr>
<td>cf-member-available?</td>
<td>422</td>
</tr>
<tr>
<td>cf-msg-checking</td>
<td>417</td>
</tr>
<tr>
<td>cf-msg-error</td>
<td>418</td>
</tr>
<tr>
<td>cf-msg-notice</td>
<td>418</td>
</tr>
<tr>
<td>cf-msg-result</td>
<td>418</td>
</tr>
<tr>
<td>cf-msg-warn</td>
<td>418</td>
</tr>
<tr>
<td>cf-output</td>
<td>425</td>
</tr>
<tr>
<td>cf-output-default</td>
<td>425</td>
</tr>
<tr>
<td>cf-output-define</td>
<td>425</td>
</tr>
<tr>
<td>cf-package-ref</td>
<td>417</td>
</tr>
<tr>
<td>cf-path-prog</td>
<td>419</td>
</tr>
<tr>
<td>cf-prog-cxx</td>
<td>420</td>
</tr>
<tr>
<td>cf-ref</td>
<td>419</td>
</tr>
<tr>
<td>cf-search-libs</td>
<td>424</td>
</tr>
<tr>
<td>cf-show-substs</td>
<td>426</td>
</tr>
<tr>
<td>cf-subst</td>
<td>419</td>
</tr>
<tr>
<td>cf-subst-append</td>
<td>419</td>
</tr>
<tr>
<td>cf-subst-prepend</td>
<td>419</td>
</tr>
<tr>
<td>cf-try-compile</td>
<td>424</td>
</tr>
<tr>
<td>cf-try-compile-and-link</td>
<td>425</td>
</tr>
<tr>
<td>cf-type-available?</td>
<td>421</td>
</tr>
<tr>
<td>cgen-add</td>
<td>387</td>
</tr>
<tr>
<td>cgen-body</td>
<td>384</td>
</tr>
<tr>
<td>cgen-box-expr</td>
<td>390</td>
</tr>
<tr>
<td>cgen-cexpr</td>
<td>387</td>
</tr>
<tr>
<td>cgen-current-unit</td>
<td>384</td>
</tr>
<tr>
<td>cgen-decl</td>
<td>384</td>
</tr>
<tr>
<td>cgen-emit-body</td>
<td>384</td>
</tr>
<tr>
<td>cgen-emit-c</td>
<td>384</td>
</tr>
<tr>
<td>cgen-emit-decl</td>
<td>387</td>
</tr>
<tr>
<td>cgen-emit-h</td>
<td>384</td>
</tr>
<tr>
<td>cgen-emit-init</td>
<td>387</td>
</tr>
<tr>
<td>cgen-emit-xtrn</td>
<td>387</td>
</tr>
<tr>
<td>cgen-external</td>
<td>384</td>
</tr>
<tr>
<td>cgen-init</td>
<td>384</td>
</tr>
<tr>
<td>cgen-literal</td>
<td>387</td>
</tr>
<tr>
<td>cgen-pred-expr</td>
<td>390</td>
</tr>
<tr>
<td>cgen-safe-comment</td>
<td>385</td>
</tr>
<tr>
<td>cgen-safe-name</td>
<td>385</td>
</tr>
<tr>
<td>cgen-safe-name-friendly</td>
<td>385</td>
</tr>
<tr>
<td>cgen-safe-string</td>
<td>385</td>
</tr>
<tr>
<td>cgen-type-from-name</td>
<td>390</td>
</tr>
<tr>
<td>cgen-unbox-expr</td>
<td>390</td>
</tr>
<tr>
<td>cgen-unit-c-file</td>
<td>384</td>
</tr>
<tr>
<td>cgen-unit-h-file</td>
<td>384</td>
</tr>
<tr>
<td>cgen-unit-init-name</td>
<td>384</td>
</tr>
<tr>
<td>cgen-with-cpp-condition</td>
<td>386</td>
</tr>
<tr>
<td>cgi-add-temporary-file</td>
<td>1050</td>
</tr>
<tr>
<td>cgi-get-metavable</td>
<td>1047</td>
</tr>
<tr>
<td>cgi-get-parameter</td>
<td>1048</td>
</tr>
<tr>
<td>cgi-header</td>
<td>1048</td>
</tr>
<tr>
<td>cgi-main</td>
<td>1049</td>
</tr>
<tr>
<td>cgi-metavariables</td>
<td>1047</td>
</tr>
<tr>
<td>cgi-output-character-encoding</td>
<td>1049</td>
</tr>
<tr>
<td>cgi-parse-parameters</td>
<td>1048</td>
</tr>
<tr>
<td>cgi-temporary-files</td>
<td>1050</td>
</tr>
<tr>
<td>cgi-test-environment-ref</td>
<td>1052</td>
</tr>
<tr>
<td>chain</td>
<td>775</td>
</tr>
<tr>
<td>chain-and</td>
<td>775</td>
</tr>
<tr>
<td>chain-lambda</td>
<td>776</td>
</tr>
<tr>
<td>chain-when</td>
<td>775</td>
</tr>
<tr>
<td>change-class</td>
<td>348</td>
</tr>
<tr>
<td>change-object-class</td>
<td>348</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td><code>char-&gt;integer</code></td>
<td>163</td>
</tr>
<tr>
<td><code>char-&gt;ucs</code></td>
<td>163</td>
</tr>
<tr>
<td><code>char-alphabetic?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-ci-hash</code></td>
<td>115</td>
</tr>
<tr>
<td><code>char-ci&lt;=?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-ci&gt;?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-ci=?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-ci&gt;=?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-ci?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-downcase</code></td>
<td>163</td>
</tr>
<tr>
<td><code>char-east-asian-width</code></td>
<td>555</td>
</tr>
<tr>
<td><code>char-foldcase</code></td>
<td>163</td>
</tr>
<tr>
<td><code>char-general-category</code></td>
<td>162</td>
</tr>
<tr>
<td><code>char-hash</code></td>
<td>115</td>
</tr>
<tr>
<td><code>char-lower-case?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-numeric?</code></td>
<td>161</td>
</tr>
<tr>
<td><code>char-ready?</code></td>
<td>272</td>
</tr>
<tr>
<td><code>char-set</code></td>
<td>170</td>
</tr>
<tr>
<td><code>char-set-&gt;list</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-&gt;ere</code></td>
<td>649</td>
</tr>
<tr>
<td><code>char-set-&gt;string</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-adjoin</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-adjoin!</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-any</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-complement</code></td>
<td>170</td>
</tr>
<tr>
<td><code>char-set-complement!</code></td>
<td>170</td>
</tr>
<tr>
<td><code>char-set-contains?</code></td>
<td>169</td>
</tr>
<tr>
<td><code>char-set-copy</code></td>
<td>170</td>
</tr>
<tr>
<td><code>char-set-count</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-cursor</code></td>
<td>615</td>
</tr>
<tr>
<td><code>char-set-cursor-next</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-delete</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-delete!</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-diff+intersection</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-diff+intersection!</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-difference</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-difference!</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-every</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-filter</code></td>
<td>614</td>
</tr>
<tr>
<td><code>char-set-fold</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-for-each</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-hash</code></td>
<td>615</td>
</tr>
<tr>
<td><code>char-set-immutable?</code></td>
<td>169</td>
</tr>
<tr>
<td><code>char-set-intersection</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-intersection!</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-map</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-ref</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-size</code></td>
<td>170</td>
</tr>
<tr>
<td><code>char-set-unfold</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-unfold!</code></td>
<td>616</td>
</tr>
<tr>
<td><code>char-set-union</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-xor</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set-xor!</code></td>
<td>617</td>
</tr>
<tr>
<td><code>char-set=</code>, <code>char=?</code>, <code>char&lt;=&gt;</code>, <code>char=&gt;</code></td>
<td>161</td>
</tr>
<tr>
<td><code>chars$</code></td>
<td>842</td>
</tr>
<tr>
<td><code>check</code></td>
<td>725</td>
</tr>
<tr>
<td><code>check=directory-tree</code></td>
<td>884</td>
</tr>
<tr>
<td><code>check-ec</code></td>
<td>725</td>
</tr>
<tr>
<td><code>check-passed?</code></td>
<td>726</td>
</tr>
<tr>
<td><code>check-report</code></td>
<td>726</td>
</tr>
<tr>
<td><code>check-reset!</code></td>
<td>726</td>
</tr>
<tr>
<td><code>check-set-mode!</code></td>
<td>726</td>
</tr>
<tr>
<td><code>check-substring-spec</code></td>
<td>701</td>
</tr>
<tr>
<td><code>chibi-test</code></td>
<td>814</td>
</tr>
<tr>
<td><code>chready?</code></td>
<td>987</td>
</tr>
<tr>
<td><code>circular-generator</code></td>
<td>436</td>
</tr>
<tr>
<td><code>circular-list</code></td>
<td>593</td>
</tr>
<tr>
<td><code>circular-list?</code></td>
<td>142</td>
</tr>
<tr>
<td><code>cise-ambient-copy</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-ambient-decl-strings</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-default-ambient</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-lookup-macro</code></td>
<td>396</td>
</tr>
<tr>
<td><code>cise-register-macro!</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-render</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-render-rec</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-render-to-string</code></td>
<td>395</td>
</tr>
<tr>
<td><code>cise-translate</code></td>
<td>395</td>
</tr>
<tr>
<td><code>clamp</code></td>
<td>131</td>
</tr>
<tr>
<td><code>class-direct-methods</code></td>
<td>341</td>
</tr>
<tr>
<td><code>class-direct-slots</code></td>
<td>341</td>
</tr>
<tr>
<td><code>class-direct-subclasses</code></td>
<td>341</td>
</tr>
<tr>
<td><code>class-direct-supers</code></td>
<td>341</td>
</tr>
<tr>
<td><code>class-name</code></td>
<td>340</td>
</tr>
<tr>
<td><code>class-of</code></td>
<td>107</td>
</tr>
<tr>
<td><code>class-post-initialize</code></td>
<td>354</td>
</tr>
<tr>
<td><code>class-precedence-list</code></td>
<td>341</td>
</tr>
<tr>
<td><code>class-redefinition</code></td>
<td>358</td>
</tr>
<tr>
<td><code>class-slot-accessor</code></td>
<td>347</td>
</tr>
<tr>
<td><code>class-slot-bound?</code></td>
<td>347</td>
</tr>
<tr>
<td><code>class-slot-definition</code></td>
<td>341</td>
</tr>
<tr>
<td><code>class-slot-ref</code></td>
<td>347</td>
</tr>
<tr>
<td><code>class-slot-set!</code></td>
<td>347</td>
</tr>
<tr>
<td><code>class-slots</code></td>
<td>341</td>
</tr>
<tr>
<td><code>clear-screen</code></td>
<td>987</td>
</tr>
<tr>
<td><code>clear-to-eol</code></td>
<td>987</td>
</tr>
<tr>
<td><code>clear-to-eos</code></td>
<td>987</td>
</tr>
<tr>
<td><code>close-directory</code></td>
<td>747</td>
</tr>
<tr>
<td><code>close-input-port</code></td>
<td>261</td>
</tr>
<tr>
<td><code>close-output-port</code></td>
<td>261</td>
</tr>
<tr>
<td><code>close-port</code></td>
<td>261</td>
</tr>
<tr>
<td><code>code</code></td>
<td>785</td>
</tr>
<tr>
<td><code>codepoints-&gt;grapheme-clusters</code></td>
<td>553</td>
</tr>
<tr>
<td><code>codepoints-&gt;words</code></td>
<td>553</td>
</tr>
<tr>
<td><code>codepoints-downcase</code></td>
<td>554</td>
</tr>
<tr>
<td><code>codepoints-foldcase</code></td>
<td>554</td>
</tr>
<tr>
<td><code>codepoints-titlecase</code></td>
<td>554</td>
</tr>
<tr>
<td><code>codepoints-upcase</code></td>
<td>554</td>
</tr>
<tr>
<td><code>coerce-to</code></td>
<td>410</td>
</tr>
<tr>
<td><code>columnar</code></td>
<td>685</td>
</tr>
<tr>
<td><code>combinations</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>combinations+*</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>combinations+*-&gt;for-each</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>combinations-*</code></td>
<td>1016</td>
</tr>
</tbody>
</table>
combinations-of.................................................. 844
combine-hash-value............................................. 115
command-args.................................................... 773
command-line..................................................... 294
command-name.................................................... 773
common-prefix.................................................... 516
common-prefix-to................................................ 516
comparator-check-type.......................................... 119
comparator-compare.............................................. 119
comparator-comparison-procedure............................ 119
comparator-comparison-procedure?........................... 732
comparator-equal?................................................ 732
comparator-equality-predicate................................. 118
comparator-flavor................................................. 118
comparator-hash.................................................. 119
comparator-hash-function....................................... 118
comparator-hash-function?..................................... 732
comparator-hashable?............................................ 118
comparator-max................................................... 744
comparator-min................................................... 744
comparator-min-in-list......................................... 744
comparator-ordered?............................................. 118
comparator-ordering-predicate................................. 118
comparator-register-default!.................................... 120
comparator-test-type............................................. 119
comparator-type-test-procedure............................... 118, 732
comparator?....................................................... 712
cmpare............................................................ 112
complement......................................................... 221
complex?................................................................ 457
component?................................................................ 123
compose............................................................... 221
compose-left......................................................... 786
compose-right....................................................... 786
compound-access.................................................. 795
compound-filter.................................................... 795
compound-length................................................... 794
compound-map....................................................... 794
compound-map->list............................................... 794
compound-predicate................................................ 795
compound-ref....................................................... 794
compound-subobjects............................................. 794
compound?.................................................................. 794
compute-cpl......................................................... 354
compute-get-n-set.................................................. 354, 356
compute-slot-accessor.............................................. 355
concatenate.......................................................... 354, 356
concatenate!......................................................... 151
concurrent-modification-violation?............................ 545
cond................................................................. 55, 392
cond-expand.......................................................... 75
cond-list............................................................. 143
condition............................................................. 258
condition-has-type?............................................... 257
condition-message.................................................. 257
condition-ref........................................................ 257
condition-type?..................................................... 257
condition-variable-broadcast!................................. 539
condition-variable-name......................................... 539
condition-variable-signal!....................................... 539
condition-variable-specific..................................... 539
condition-variable-specific-set!............................... 539
condition-variable?............................................... 539
condition?.......................................................... 257
continue.............................................................. 393
counter.................................................................... 150
count$................................................................. 220
count-columns........................................................ 633
count-accumulator.................................................. 633
cpu-architecture.................................................... 730
crc32................................................................. 959
creates-newfile..................................................... 746
creates-symlink.................................................... 746
creates-temps-file.................................................. 748
csv-rows->tuples.................................................... 989
current-class-of.................................................... 347
current-classification.............................................. 347
current-continuation-mark...................................... 229
current-continuation.............................................. 708
current-date.......................................................... 702
current-dates........................................................ 702
current-directories............................................... 881
current-directory.................................................. 881
current-directory-path.......................................... 881
current-directory-path-name.................................... 881
current-directory-root.......................................... 881
current-dynamic-extent.......................................... 743
current-error-port................................................ 260
current-exception................................................ 252
current-exception-handler...................................... 252
current-file-descriptor.......................................... 260
current-file-descriptor-name................................... 260
current-get-n-set.................................................. 260
current-get-n-set-name.......................................... 260
current-jiffy.......................................................... 591
current-julian-day................................................ 702
continue.............................................................. 393
counter.................................................................... 150
count$................................................................. 220
count-columns........................................................ 633
count-accumulator.................................................. 633
cpu-architecture.................................................... 730
crc32................................................................. 959
creates-newfile..................................................... 746
creates-symlink.................................................... 746
creates-temps-file.................................................. 748
csv-rows->tuples.................................................... 989
current-class-of.................................................... 347
current-classification.............................................. 347
current-continuation-mark...................................... 229
current-continuation.............................................. 708
current-date.......................................................... 702
current-dates........................................................ 702
current-directories............................................... 881
current-directory.................................................. 881
current-directory-path.......................................... 881
current-directory-path-name................................... 881
current-directory-root.......................................... 881
current-dynamic-extent.......................................... 743
current-error-port................................................ 260
current-exception................................................ 252
current-exception-handler...................................... 252
current-file-descriptor.......................................... 260
current-file-descriptor-name................................... 260
current-get-n-set.................................................. 260
current-get-n-set-name.......................................... 260
current-jiffy.......................................................... 591
current-julian-day................................................ 702
### Appendix B: Function and Syntax Index

<table>
<thead>
<tr>
<th>Function/Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>current-language</td>
<td>708</td>
</tr>
<tr>
<td>current-load-history</td>
<td>287</td>
</tr>
<tr>
<td>current-load-next</td>
<td>287</td>
</tr>
<tr>
<td>current-load-path</td>
<td>287</td>
</tr>
<tr>
<td>current-load-port</td>
<td>287</td>
</tr>
<tr>
<td>current-locale-details</td>
<td>708</td>
</tr>
<tr>
<td>current-modified-julian-day</td>
<td>702</td>
</tr>
<tr>
<td>current-module</td>
<td>81</td>
</tr>
<tr>
<td>current-output-port</td>
<td>260</td>
</tr>
<tr>
<td>current-parameterization</td>
<td>235</td>
</tr>
<tr>
<td>current-second</td>
<td>591</td>
</tr>
<tr>
<td>current-thread</td>
<td>533</td>
</tr>
<tr>
<td>current-time</td>
<td>318, 702</td>
</tr>
<tr>
<td>current-trace-port</td>
<td>260</td>
</tr>
<tr>
<td>curried</td>
<td>804</td>
</tr>
<tr>
<td>cursor-down/scroll-up</td>
<td>988</td>
</tr>
<tr>
<td>cursor-up/scroll-down</td>
<td>988</td>
</tr>
<tr>
<td>cut</td>
<td>49</td>
</tr>
<tr>
<td>cute</td>
<td>49</td>
</tr>
</tbody>
</table>

### D

<table>
<thead>
<tr>
<th>Function/Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>449</td>
</tr>
<tr>
<td>date-&gt;julian-day</td>
<td>704</td>
</tr>
<tr>
<td>date-&gt;modified-julian-day</td>
<td>704</td>
</tr>
<tr>
<td>date-&gt;rfc222-date</td>
<td>921</td>
</tr>
<tr>
<td>date-&gt;string</td>
<td>705</td>
</tr>
<tr>
<td>date-&gt;time-monotonic</td>
<td>704</td>
</tr>
<tr>
<td>date-&gt;time-tai</td>
<td>704</td>
</tr>
<tr>
<td>date-&gt;time-utc</td>
<td>704</td>
</tr>
<tr>
<td>date-day</td>
<td>704</td>
</tr>
<tr>
<td>date-hour</td>
<td>704</td>
</tr>
<tr>
<td>date-minute</td>
<td>704</td>
</tr>
<tr>
<td>date-month</td>
<td>704</td>
</tr>
<tr>
<td>date-nanosecond</td>
<td>704</td>
</tr>
<tr>
<td>date-second</td>
<td>704</td>
</tr>
<tr>
<td>date-week-day</td>
<td>704</td>
</tr>
<tr>
<td>date-week-number</td>
<td>704</td>
</tr>
<tr>
<td>date-year</td>
<td>704</td>
</tr>
<tr>
<td>date-year-day</td>
<td>704</td>
</tr>
<tr>
<td>date-zone-offset</td>
<td>704</td>
</tr>
<tr>
<td>date?</td>
<td>704</td>
</tr>
<tr>
<td>dbi-close</td>
<td>867, 868, 869, 870</td>
</tr>
<tr>
<td>dbi-connect</td>
<td>866</td>
</tr>
<tr>
<td>dbi-do</td>
<td>868, 870</td>
</tr>
<tr>
<td>dbi-escape-sql</td>
<td>868, 870</td>
</tr>
<tr>
<td>dbi-execute</td>
<td>868</td>
</tr>
<tr>
<td>dbi-execute-using-connection</td>
<td>870</td>
</tr>
<tr>
<td>dbi-list-drivers</td>
<td>867</td>
</tr>
<tr>
<td>dbi-make-connection</td>
<td>869</td>
</tr>
<tr>
<td>dbi-make-driver</td>
<td>867</td>
</tr>
<tr>
<td>dbi-open?</td>
<td>866, 868, 870</td>
</tr>
<tr>
<td>dbi-parse-dm</td>
<td>870</td>
</tr>
<tr>
<td>dbi-prepare</td>
<td>867, 870</td>
</tr>
<tr>
<td>dbi-prepare-sql</td>
<td>871</td>
</tr>
<tr>
<td>dbm-close</td>
<td>873</td>
</tr>
<tr>
<td>dbm-closed?</td>
<td>873</td>
</tr>
<tr>
<td>dbm-db-copy</td>
<td>874</td>
</tr>
<tr>
<td>dbm-db-exists?</td>
<td>874</td>
</tr>
<tr>
<td>dbm-db-move</td>
<td>875</td>
</tr>
<tr>
<td>dbm-db-remove</td>
<td>874</td>
</tr>
<tr>
<td>dbm-delete!</td>
<td>874</td>
</tr>
<tr>
<td>dbm-exists?</td>
<td>874</td>
</tr>
<tr>
<td>dbm-fold</td>
<td>874</td>
</tr>
<tr>
<td>dbm-for-each</td>
<td>874</td>
</tr>
<tr>
<td>dbm-get</td>
<td>874</td>
</tr>
<tr>
<td>dbm-map</td>
<td>874</td>
</tr>
<tr>
<td>dbm-open</td>
<td>873</td>
</tr>
<tr>
<td>dbm-put!</td>
<td>873</td>
</tr>
<tr>
<td>dbm-type-&gt;class</td>
<td>873</td>
</tr>
<tr>
<td>dgettext</td>
<td>1001</td>
</tr>
<tr>
<td>debug-funcall</td>
<td>326</td>
</tr>
<tr>
<td>debug-funcall-conditionally</td>
<td>326</td>
</tr>
<tr>
<td>debug-label</td>
<td>327</td>
</tr>
<tr>
<td>debug-print</td>
<td>326</td>
</tr>
<tr>
<td>debug-print-conditionally</td>
<td>326</td>
</tr>
<tr>
<td>debug-print-width</td>
<td>326</td>
</tr>
<tr>
<td>debug-source-info</td>
<td>326</td>
</tr>
<tr>
<td>dec!</td>
<td>54</td>
</tr>
<tr>
<td>declare-bundle!</td>
<td>708</td>
</tr>
<tr>
<td>declare-cfn</td>
<td>394, 399</td>
</tr>
<tr>
<td>declare-cvar</td>
<td>394, 399</td>
</tr>
<tr>
<td>decode</td>
<td>399</td>
</tr>
<tr>
<td>decode-float</td>
<td>134</td>
</tr>
<tr>
<td>decompose-path</td>
<td>885</td>
</tr>
<tr>
<td>default-continuation-prompt-tag</td>
<td>228</td>
</tr>
<tr>
<td>default-endian</td>
<td>139</td>
</tr>
<tr>
<td>default-file-encoding</td>
<td>267</td>
</tr>
<tr>
<td>default-hash</td>
<td>113</td>
</tr>
<tr>
<td>default-mapper</td>
<td>823</td>
</tr>
<tr>
<td>default-sizer</td>
<td>845</td>
</tr>
<tr>
<td>default-tls-class</td>
<td>948</td>
</tr>
<tr>
<td>define</td>
<td>68</td>
</tr>
<tr>
<td>define-cclass</td>
<td>397</td>
</tr>
<tr>
<td>define-cfn</td>
<td>393, 399</td>
</tr>
<tr>
<td>define-cc</td>
<td>397</td>
</tr>
<tr>
<td>define-cc</td>
<td>396, 399</td>
</tr>
<tr>
<td>define-cc</td>
<td>396</td>
</tr>
<tr>
<td>define-courier</td>
<td>396</td>
</tr>
<tr>
<td>define-courier</td>
<td>396</td>
</tr>
<tr>
<td>define-cc-toplevel</td>
<td>396</td>
</tr>
<tr>
<td>define-class</td>
<td>336</td>
</tr>
<tr>
<td>define-cmethod</td>
<td>397</td>
</tr>
<tr>
<td>define-condition-type</td>
<td>256</td>
</tr>
<tr>
<td>define-constant</td>
<td>70, 398</td>
</tr>
<tr>
<td>define-cproc</td>
<td>396</td>
</tr>
<tr>
<td>define-cptr</td>
<td>398</td>
</tr>
<tr>
<td>define-c-type</td>
<td>394, 399</td>
</tr>
<tr>
<td>define-curried</td>
<td>804</td>
</tr>
<tr>
<td>define-cvar</td>
<td>394, 399</td>
</tr>
<tr>
<td>define-dict-interface</td>
<td>429</td>
</tr>
<tr>
<td>define-enum</td>
<td>399, 779</td>
</tr>
<tr>
<td>define-enum-conditionally</td>
<td>399</td>
</tr>
<tr>
<td>define-enumeration</td>
<td>784</td>
</tr>
<tr>
<td>define-gauche-package</td>
<td>478</td>
</tr>
<tr>
<td>define-generic</td>
<td>349</td>
</tr>
<tr>
<td>define-hybrid-syntax</td>
<td>98</td>
</tr>
<tr>
<td>define-in-module</td>
<td>72</td>
</tr>
<tr>
<td>define-inline</td>
<td>71</td>
</tr>
<tr>
<td>define-library</td>
<td>584</td>
</tr>
<tr>
<td>define-macro</td>
<td>97</td>
</tr>
<tr>
<td>define-method</td>
<td>349</td>
</tr>
<tr>
<td>define-module</td>
<td>81</td>
</tr>
<tr>
<td>define-options</td>
<td>801</td>
</tr>
<tr>
<td>define-options*</td>
<td>801</td>
</tr>
<tr>
<td>define-reader-ctor</td>
<td>273</td>
</tr>
<tr>
<td>define-record-type</td>
<td>503</td>
</tr>
<tr>
<td>define-stream</td>
<td>1037</td>
</tr>
<tr>
<td>define-symbol</td>
<td>398</td>
</tr>
<tr>
<td>define-syntax</td>
<td>90</td>
</tr>
<tr>
<td>define-type</td>
<td>396</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>define-values</td>
<td>70</td>
</tr>
<tr>
<td>define-variable</td>
<td>398</td>
</tr>
<tr>
<td>deflate-string</td>
<td>958</td>
</tr>
<tr>
<td>deflating-port-full-flush</td>
<td>958</td>
</tr>
<tr>
<td>degrees-&gt;radians</td>
<td>132</td>
</tr>
<tr>
<td>delay</td>
<td>239</td>
</tr>
<tr>
<td>delay-force</td>
<td>590</td>
</tr>
<tr>
<td>delete</td>
<td>150</td>
</tr>
<tr>
<td>delete!</td>
<td>150</td>
</tr>
<tr>
<td>delete$</td>
<td>221</td>
</tr>
<tr>
<td>delete-directory</td>
<td>746</td>
</tr>
<tr>
<td>delete-directory*</td>
<td>883</td>
</tr>
<tr>
<td>delete-duplicates</td>
<td>150</td>
</tr>
<tr>
<td>delete-duplicates!</td>
<td>150</td>
</tr>
<tr>
<td>delete-environment-variable!</td>
<td>749</td>
</tr>
<tr>
<td>delete-file</td>
<td>889</td>
</tr>
<tr>
<td>delete-files</td>
<td>889</td>
</tr>
<tr>
<td>delete-hook!</td>
<td>448</td>
</tr>
<tr>
<td>delete-keyword</td>
<td>158</td>
</tr>
<tr>
<td>delete-keyword!</td>
<td>158</td>
</tr>
<tr>
<td>delete-keywords</td>
<td>158</td>
</tr>
<tr>
<td>delete-keywords!</td>
<td>158</td>
</tr>
<tr>
<td>delete-neighbor-dups</td>
<td>515</td>
</tr>
<tr>
<td>delete-neighbor-dups!</td>
<td>515</td>
</tr>
<tr>
<td>delete-neighbor-dups-squeeze!</td>
<td>516</td>
</tr>
<tr>
<td>denominator</td>
<td>139</td>
</tr>
<tr>
<td>dequeue!</td>
<td>839</td>
</tr>
<tr>
<td>dequeue-all!</td>
<td>839</td>
</tr>
<tr>
<td>dequeue/wait!</td>
<td>840</td>
</tr>
<tr>
<td>describe</td>
<td>449</td>
</tr>
<tr>
<td>determinant</td>
<td>372</td>
</tr>
<tr>
<td>determinant!</td>
<td>372</td>
</tr>
<tr>
<td>dgetext</td>
<td>1001</td>
</tr>
<tr>
<td>dict-&gt;alist</td>
<td>429</td>
</tr>
<tr>
<td>dict-clear!</td>
<td>428</td>
</tr>
<tr>
<td>dict-comparator</td>
<td>428</td>
</tr>
<tr>
<td>dict-delete!</td>
<td>428</td>
</tr>
<tr>
<td>dict-exists?</td>
<td>429</td>
</tr>
<tr>
<td>dict-fold</td>
<td>429</td>
</tr>
<tr>
<td>dict-fold-right</td>
<td>429</td>
</tr>
<tr>
<td>dict-for-each</td>
<td>429</td>
</tr>
<tr>
<td>dict-get</td>
<td>428</td>
</tr>
<tr>
<td>dict-keys</td>
<td>429</td>
</tr>
<tr>
<td>dict-map</td>
<td>429</td>
</tr>
<tr>
<td>dict-pop!</td>
<td>429</td>
</tr>
<tr>
<td>dict-push!</td>
<td>429</td>
</tr>
<tr>
<td>dict-put!</td>
<td>428</td>
</tr>
<tr>
<td>dict-update!</td>
<td>429</td>
</tr>
<tr>
<td>dict-values</td>
<td>429</td>
</tr>
<tr>
<td>diff</td>
<td>991</td>
</tr>
<tr>
<td>diff-report</td>
<td>992</td>
</tr>
<tr>
<td>diff-report/context</td>
<td>993</td>
</tr>
<tr>
<td>digest</td>
<td>1018</td>
</tr>
<tr>
<td>digest-final!</td>
<td>1018</td>
</tr>
<tr>
<td>digest-hexify</td>
<td>1018</td>
</tr>
<tr>
<td>digest-message-to</td>
<td>1017</td>
</tr>
<tr>
<td>digest-string</td>
<td>1018</td>
</tr>
<tr>
<td>digest-to</td>
<td>1017</td>
</tr>
<tr>
<td>digest-update!</td>
<td>1018</td>
</tr>
<tr>
<td>digit-&gt;integer</td>
<td>164</td>
</tr>
<tr>
<td>digit-value</td>
<td>588</td>
</tr>
<tr>
<td>directory-files</td>
<td>747</td>
</tr>
<tr>
<td>directory-fold</td>
<td>882</td>
</tr>
<tr>
<td>directory-list</td>
<td>882</td>
</tr>
<tr>
<td>directory-list2</td>
<td>882</td>
</tr>
<tr>
<td>disasm</td>
<td>327</td>
</tr>
<tr>
<td>disjoint</td>
<td>805</td>
</tr>
<tr>
<td>display</td>
<td>278</td>
</tr>
<tr>
<td>display-multicolumn</td>
<td>1006</td>
</tr>
<tr>
<td>display/page</td>
<td>1007</td>
</tr>
<tr>
<td>displayed</td>
<td>682</td>
</tr>
<tr>
<td>div</td>
<td>128</td>
</tr>
<tr>
<td>div-and-mod</td>
<td>128</td>
</tr>
<tr>
<td>div0</td>
<td>128</td>
</tr>
<tr>
<td>div0-and-mod0</td>
<td>128</td>
</tr>
<tr>
<td>dl-distance</td>
<td>1023</td>
</tr>
<tr>
<td>dl-distances</td>
<td>1023</td>
</tr>
<tr>
<td>do</td>
<td>63</td>
</tr>
<tr>
<td>do-ec</td>
<td>712</td>
</tr>
<tr>
<td>do-generator</td>
<td>447</td>
</tr>
<tr>
<td>do-pipeline</td>
<td>494</td>
</tr>
<tr>
<td>do-process</td>
<td>488</td>
</tr>
<tr>
<td>do-process!</td>
<td>488</td>
</tr>
<tr>
<td>dolist</td>
<td>65, 393</td>
</tr>
<tr>
<td>dopairs</td>
<td>393</td>
</tr>
<tr>
<td>dotimes</td>
<td>65, 393</td>
</tr>
<tr>
<td>dotted-iplist?</td>
<td>621</td>
</tr>
<tr>
<td>dotted-list?</td>
<td>142</td>
</tr>
<tr>
<td>drop</td>
<td>145</td>
</tr>
<tr>
<td>drop*</td>
<td>145</td>
</tr>
<tr>
<td>drop-right</td>
<td>145</td>
</tr>
<tr>
<td>drop-right!</td>
<td>146</td>
</tr>
<tr>
<td>drop-right*</td>
<td>146</td>
</tr>
<tr>
<td>drop-while</td>
<td>596</td>
</tr>
<tr>
<td>dynamic-extent?</td>
<td>743</td>
</tr>
<tr>
<td>dynamic-lambda</td>
<td>743</td>
</tr>
<tr>
<td>dynamic-load</td>
<td>287</td>
</tr>
<tr>
<td>dynamic-wind</td>
<td>226</td>
</tr>
</tbody>
</table>

E

each | 684

each-in-list | 684

each-of | 805
eager | 240
eager-and-procedure | 806

eager-or-procedure | 806
ecase | 56
ed | 450, 997
ed-pick-file | 997
ed-string | 997
edn-equal? | 995
edn-map | 995
edn-object-handler | 996
edn-object-payload | 996
edn-object-tag | 996
edn-object? | 996
edn-set | 995
edn-symbol-basename | 996
edn-symbol-prefix | 996
edn-valid-symbol-name? | 996
edn-write | 996
eighth | 594
either->generation | 770
either->list | 769
either->list-truth | 770
either->maybe | 766
either->truth | 769
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>either-&gt;values</td>
<td>770</td>
</tr>
<tr>
<td>either-and</td>
<td>771</td>
</tr>
<tr>
<td>either-compose</td>
<td>767</td>
</tr>
<tr>
<td>either-filter</td>
<td>768</td>
</tr>
<tr>
<td>either-fold</td>
<td>768</td>
</tr>
<tr>
<td>either-for-each</td>
<td>768</td>
</tr>
<tr>
<td>either-guard</td>
<td>772</td>
</tr>
<tr>
<td>either-join</td>
<td>767</td>
</tr>
<tr>
<td>either-length</td>
<td>767</td>
</tr>
<tr>
<td>either-let*</td>
<td>771</td>
</tr>
<tr>
<td>either-let+values</td>
<td>771</td>
</tr>
<tr>
<td>either-map</td>
<td>768</td>
</tr>
<tr>
<td>either-or</td>
<td>771</td>
</tr>
<tr>
<td>either-ref</td>
<td>766</td>
</tr>
<tr>
<td>either-ref/default</td>
<td>767</td>
</tr>
<tr>
<td>either-remove</td>
<td>768</td>
</tr>
<tr>
<td>either-sequence</td>
<td>768</td>
</tr>
<tr>
<td>either-swap</td>
<td>766</td>
</tr>
<tr>
<td>either-ufold</td>
<td>768</td>
</tr>
<tr>
<td>either-when</td>
<td>766</td>
</tr>
<tr>
<td>either?</td>
<td>765</td>
</tr>
<tr>
<td>emergency-exit</td>
<td>591</td>
</tr>
<tr>
<td>end-of-char-set?</td>
<td>616</td>
</tr>
<tr>
<td>endianness</td>
<td>679</td>
</tr>
<tr>
<td>enqueue!</td>
<td>839</td>
</tr>
<tr>
<td>enqueue-unique!</td>
<td>840</td>
</tr>
<tr>
<td>enum-empty-set</td>
<td>782</td>
</tr>
<tr>
<td>enum-max</td>
<td>780</td>
</tr>
<tr>
<td>enum-min</td>
<td>780</td>
</tr>
<tr>
<td>enum-name</td>
<td>781</td>
</tr>
<tr>
<td>enum-name-&gt;enum</td>
<td>781</td>
</tr>
<tr>
<td>enum-name-&gt;ordinal</td>
<td>781</td>
</tr>
<tr>
<td>enum-name-&gt;value</td>
<td>781</td>
</tr>
<tr>
<td>enum-next</td>
<td>781</td>
</tr>
<tr>
<td>enum-ordinal</td>
<td>781</td>
</tr>
<tr>
<td>enum-ordinal-&gt;enum</td>
<td>781</td>
</tr>
<tr>
<td>enum-ordinal-&gt;name</td>
<td>781</td>
</tr>
<tr>
<td>enum-ordinal-&gt;value</td>
<td>781</td>
</tr>
<tr>
<td>enum-prev</td>
<td>782</td>
</tr>
<tr>
<td>enum-set</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-&gt;enum-list</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-&gt;list</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-&gt;adjoint</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-&gt;adjoint!</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-&gt;any?</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-complement</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-complement!</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-contains?</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-copy</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-constructor</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-count</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-delete</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-delete!</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-delete-all</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-delete-all!</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-difference</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-difference!</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-disjoint?</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-empty?</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-every?</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-filter</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-fold</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-for-each</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-indexer</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-intersection</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-intersection!</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-map-&gt;list</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-member?</td>
<td>784</td>
</tr>
<tr>
<td>enum-set-projection</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-remove</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-size</td>
<td>783</td>
</tr>
<tr>
<td>enum-set-subset?</td>
<td>782</td>
</tr>
<tr>
<td>enum-set-type</td>
<td>783</td>
</tr>
<tr>
<td>enum-set→enum-set</td>
<td>782</td>
</tr>
<tr>
<td>enum-set→contains?</td>
<td>781</td>
</tr>
<tr>
<td>enum-set→enums</td>
<td>780</td>
</tr>
<tr>
<td>enum-set→names</td>
<td>780</td>
</tr>
<tr>
<td>enum-type</td>
<td>781</td>
</tr>
<tr>
<td>enum-type→enum-set</td>
<td>782</td>
</tr>
<tr>
<td>enum-type→contains?</td>
<td>781</td>
</tr>
<tr>
<td>enum-type→enums</td>
<td>780</td>
</tr>
<tr>
<td>enum-type→names</td>
<td>780</td>
</tr>
<tr>
<td>enum-type-size</td>
<td>780</td>
</tr>
<tr>
<td>enum-type-values</td>
<td>780</td>
</tr>
<tr>
<td>enum-value</td>
<td>781</td>
</tr>
<tr>
<td>enum&lt;=?</td>
<td>781</td>
</tr>
<tr>
<td>enum&lt;=?</td>
<td>781</td>
</tr>
<tr>
<td>enum=&gt;?</td>
<td>781</td>
</tr>
<tr>
<td>enum=&gt;?</td>
<td>781</td>
</tr>
<tr>
<td>enum?</td>
<td>781</td>
</tr>
<tr>
<td>environment</td>
<td>589</td>
</tr>
<tr>
<td>eof-object</td>
<td>272</td>
</tr>
<tr>
<td>eof-object?</td>
<td>272</td>
</tr>
<tr>
<td>ephemeron-broken?</td>
<td>640</td>
</tr>
<tr>
<td>ephemeron-datum</td>
<td>640</td>
</tr>
<tr>
<td>ephemeron-key</td>
<td>640</td>
</tr>
<tr>
<td>ephemeron?</td>
<td>640</td>
</tr>
<tr>
<td>eq-compare</td>
<td>113</td>
</tr>
<tr>
<td>eq-hash</td>
<td>113</td>
</tr>
<tr>
<td>eq?</td>
<td>110</td>
</tr>
<tr>
<td>equal?</td>
<td>111</td>
</tr>
<tr>
<td>eqv-hash</td>
<td>113</td>
</tr>
<tr>
<td>eqv?</td>
<td>110</td>
</tr>
<tr>
<td>er-macro-transformer</td>
<td>93</td>
</tr>
<tr>
<td>error</td>
<td>248</td>
</tr>
<tr>
<td>error-object-irritants</td>
<td>587</td>
</tr>
<tr>
<td>error-object-message</td>
<td>587</td>
</tr>
<tr>
<td>error-object-?</td>
<td>587</td>
</tr>
<tr>
<td>error?</td>
<td>255</td>
</tr>
<tr>
<td>errof</td>
<td>249</td>
</tr>
<tr>
<td>escaped</td>
<td>682</td>
</tr>
<tr>
<td>stypecase</td>
<td>57</td>
</tr>
<tr>
<td>euclidean-quotient</td>
<td>664</td>
</tr>
<tr>
<td>euclidean-remainder</td>
<td>664</td>
</tr>
<tr>
<td>euclidean/</td>
<td>664</td>
</tr>
<tr>
<td>eval</td>
<td>258, 589</td>
</tr>
<tr>
<td>even?</td>
<td>124</td>
</tr>
<tr>
<td>every</td>
<td>150</td>
</tr>
</tbody>
</table>
every$ .............................................. 221
every-bit-set? .................................... 666
every-in-queue .............................. 840
every-pred ............................ 222
every?-ec ...................................... 713
exact .............................................. 135
exact->inexact .................................. 135
exact-integer-sqrt ....................... 132
exact-integer? ............................. 124
exact? .......................................... 124
exception->either ................................. 771
exception-handler-stack ................. 252
exit ............................... 293
exit-handler .................................. 294
exp .............................................. 131
expand-path .................................... 884
expand-template-file ...................... 1012
expand-template-string .................. 1012
export ............................. 81
export-all .................................... 81
expt .......................................... 132
expt-mod ............................... 132
extend ......................................... 83
extended-cons .................................. 154
extended-list .................................. 154
extended-pair? ................................. 154
external-conversion-library ............. 400
extract-condition ............................ 257
f16?
............... 557
f16array ...................................... 368
f16ite-u8vector .............................. 568
f16ke-u8vector-generator ............... 562
f16vector ................................... 557
f16vector->list ............................. 563
f16vector->vector ........................... 564
f16vector-ADJAX ................................ 564
f16vector-ADD ................................ 565
f16vector-any .................................. 562
f16vector-append ................................ 561
f16vector-APPEND-SUBVECTORS ........... 561
f16vector-clamp! ............................ 567
f16vector-compare ............................ 558
f16vector-CONCATENATE .................... 561
f16vector-Copy .............................. 559
f16vector-Copy! ............................. 559
f16vector-COUNT ........................... 562
f16vector-CUMULATE ......................... 562
f16vector-DIV! ............................... 566
f16vector-dot! ............................... 566
f16vector-drop .............................. 561
f16vector-drop-right ......................... 561
f16vector-drop-while ......................... 562
f16vector-drop-while-right ................ 562
f16vector-empty? ............................. 557
f16vector-every .............................. 562
f16vector-fill! .................................. 558
f16vector-filter ............................. 562
f16vector-fold .............................. 561
f16vector-for-each .......................... 562
f16vector-index ............................. 562
f16vector-index-right ....................... 562
f16vector-length ......................... 558
f16vector-map ................................ 562
f16vector-map! .............................. 562
f16vector-mul ............................... 565
f16vector-mul! .............................. 565
f16vector-multi-copy! ....................... 559
f16vector-partition ......................... 562
f16vector-range-check ...................... 566
f16vector-ref ............................... 201
f16vector-REM O V E R .......................... 562
f16vector-reverse! .......................... 559
f16vector-reverse-copy ..................... 559
f16vector-reverse-copy! .................... 559
f16vector-segment .......................... 561
f16vector-set! .................................. 201
f16vector-skip .................................. 562
f16vector-skip-right ....................... 562
f16vector-sub! ............................... 565
f16vector-sub ............................ 565
f16vector-take .............................. 561
f16vector-take-right ......................... 561
f16vector-take-while .......................... 562
f16vector-take-while-right ................. 562
f16vector-unfold-right! ..................... 561
f16vector-unfold ............................ 557
f16vector-unfold-right ........................ 557
f16vector-unfold-right! ..................... 557
f16vector= ..................................... 558
f16vector=.................................. 558
f16vector? ..................................... 200
f32?
.................. 557
f32array ..................................... 368
f32ite-u8vector .............................. 568
f32ke-u8vector-generator ............... 562
f32vector ................................. 557
f32vector->list ............................. 563
f32vector->vector ........................... 563
f32vector-ADJAX ............................. 563
f32vector-COUNT ........................... 562
f32vector-CUMULATE ......................... 562
f32vector-DIV! ............................... 566
f32vector-dot! ............................... 566
f32vector-drop .............................. 561
f32vector-drop-right ......................... 561
f32vector-drop-while ......................... 562
f32vector-drop-while-right ................ 562
f32vector-empty? ............................. 557
f32vector-every .............................. 562
f32vector-fill! .................................. 558
f32vector-filter ............................. 562
f32vector-fold .............................. 561
f32vector-for-each .......................... 562
f32vector-index ............................. 562
f32vector-index-right ....................... 562
f32vector-length ............................. 558
f32vector-map ................................ 562
f32vector-map! .............................. 562
f32vector-mul ............................... 565
f32vector-mul! .............................. 565
f32vector-multi-copy! ....................... 559
f32vector-partition .......................... 562
f32vector-range-check ...................... 566
f32vector-ref ............................... 201
f32vector-REM O V E R .......................... 562
f32vector-reverse ............................ 559
f32vector-reverse-copy ..................... 559
f32vector-reverse-copy! .................... 559
f32vector-segment .......................... 561
f32vector-set! .................................. 201
f32vector-skip .................................. 562
f32vector-skip-right ....................... 562
f32vector-sub! ............................... 565
f32vector-sub ............................ 565
f32vector-take .............................. 561
f32vector-take-right ......................... 561
f32vector-take-while .......................... 562
f32vector-take-while-right ................ 562
f32vector-unfold-right! ..................... 561
f32vector-unfold ............................ 557
f32vector-unfold-right ........................ 557
f32vector-unfold-right! ..................... 557
f32vector=.................................. 558
f32vector=.................................. 558
f32vector? ..................................... 200
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>f32vector-index</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-length</td>
<td>558</td>
</tr>
<tr>
<td>f32vector-map</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-map!</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-mul</td>
<td>565</td>
</tr>
<tr>
<td>f32vector-mul!</td>
<td>565</td>
</tr>
<tr>
<td>f32vector-multi-copy!</td>
<td>559</td>
</tr>
<tr>
<td>f32vector-partition</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-range-check</td>
<td>566</td>
</tr>
<tr>
<td>f32vector-ref</td>
<td>201</td>
</tr>
<tr>
<td>f32vector-remove</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-reverse</td>
<td>559</td>
</tr>
<tr>
<td>f32vector-reverse-copy!</td>
<td>559</td>
</tr>
<tr>
<td>f32vector-segment</td>
<td>561</td>
</tr>
<tr>
<td>f32vector-set!</td>
<td>201</td>
</tr>
<tr>
<td>f32vector-skip</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-skip-right</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>f32vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>f32vector-swap!</td>
<td>558</td>
</tr>
<tr>
<td>f32vector-take</td>
<td>561</td>
</tr>
<tr>
<td>f32vector-take-right</td>
<td>561</td>
</tr>
<tr>
<td>f32vector-take-while</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-take-while-right</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-unfold</td>
<td>557, 561</td>
</tr>
<tr>
<td>f32vector-unfold!</td>
<td>557, 561</td>
</tr>
<tr>
<td>f32vector-unfold-right</td>
<td>557</td>
</tr>
<tr>
<td>f32vector-unfold-right!</td>
<td>557</td>
</tr>
<tr>
<td>f32vector=</td>
<td>558</td>
</tr>
<tr>
<td>f32vector?</td>
<td>558</td>
</tr>
<tr>
<td>f32vector?</td>
<td>200</td>
</tr>
<tr>
<td>f64?</td>
<td>557</td>
</tr>
<tr>
<td>f64array</td>
<td>368</td>
</tr>
<tr>
<td>f64bit-u8vector</td>
<td>568</td>
</tr>
<tr>
<td>f64ke-u8vector-generator</td>
<td>562</td>
</tr>
<tr>
<td>f64vector</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-&gt;list</td>
<td>563</td>
</tr>
<tr>
<td>f64vector-&gt;vector</td>
<td>564</td>
</tr>
<tr>
<td>f64vector-add</td>
<td>565</td>
</tr>
<tr>
<td>f64vector-add!</td>
<td>565</td>
</tr>
<tr>
<td>f64vector-any</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-append</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-append-subvectors</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-clamp</td>
<td>567</td>
</tr>
<tr>
<td>f64vector-clone</td>
<td>567</td>
</tr>
<tr>
<td>f64vector-compare</td>
<td>558</td>
</tr>
<tr>
<td>f64vector-concatenate</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-copy</td>
<td>559</td>
</tr>
<tr>
<td>f64vector-copy!</td>
<td>559</td>
</tr>
<tr>
<td>f64vector-count</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-cumulate</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-div</td>
<td>566</td>
</tr>
<tr>
<td>f64vector-div!</td>
<td>566</td>
</tr>
<tr>
<td>f64vector-dot</td>
<td>566</td>
</tr>
<tr>
<td>f64vector-dot!</td>
<td>566</td>
</tr>
<tr>
<td>f64vector-drop</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-drop-right</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-drop-while</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-drop-while-right</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-empty</td>
<td>557</td>
</tr>
<tr>
<td>f64vector-every</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-fill!</td>
<td>558</td>
</tr>
<tr>
<td>f64vector-filter</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-fold</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-for-each</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-index</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-map</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-map!</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-mul</td>
<td>565</td>
</tr>
<tr>
<td>f64vector-mul!</td>
<td>565</td>
</tr>
<tr>
<td>f64vector-multi-copy!</td>
<td>559</td>
</tr>
<tr>
<td>f64vector-partition</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-range-check</td>
<td>566</td>
</tr>
<tr>
<td>f64vector-ref</td>
<td>201</td>
</tr>
<tr>
<td>f64vector-remove</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-reverse</td>
<td>559</td>
</tr>
<tr>
<td>f64vector-reverse-copy!</td>
<td>559</td>
</tr>
<tr>
<td>f64vector-reverse-copy!</td>
<td>559</td>
</tr>
<tr>
<td>f64vector-segment</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-set!</td>
<td>201</td>
</tr>
<tr>
<td>f64vector-skip</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-skip-right</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>f64vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>f64vector-swap!</td>
<td>558</td>
</tr>
<tr>
<td>f64vector-take</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-take-right</td>
<td>561</td>
</tr>
<tr>
<td>f64vector-take-while</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-take-while-right</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-unfold</td>
<td>557, 561</td>
</tr>
<tr>
<td>f64vector-unfold!</td>
<td>557, 561</td>
</tr>
<tr>
<td>f64vector-unfold-right</td>
<td>557</td>
</tr>
<tr>
<td>f64vector-unfold-right!</td>
<td>557</td>
</tr>
<tr>
<td>f64vector=</td>
<td>558</td>
</tr>
<tr>
<td>f64vector=</td>
<td>558</td>
</tr>
<tr>
<td>f64vector?</td>
<td>200</td>
</tr>
<tr>
<td>f64vector?</td>
<td>558</td>
</tr>
<tr>
<td>f64vector?</td>
<td>558</td>
</tr>
<tr>
<td>f64vector?</td>
<td>558</td>
</tr>
<tr>
<td>f64vector?</td>
<td>200</td>
</tr>
<tr>
<td>f64vector?</td>
<td>558</td>
</tr>
<tr>
<td>f64vector?</td>
<td>558</td>
</tr>
<tr>
<td>f64vector?</td>
<td>558</td>
</tr>
<tr>
<td>Function Description</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>find-module</td>
<td>587</td>
</tr>
<tr>
<td>find-string-from-port</td>
<td>1007</td>
</tr>
<tr>
<td>file-error</td>
<td>301</td>
</tr>
<tr>
<td>file-exists</td>
<td>880</td>
</tr>
<tr>
<td>file-filter</td>
<td>881</td>
</tr>
<tr>
<td>file-filter-fold</td>
<td>881</td>
</tr>
<tr>
<td>file-filter-for-each</td>
<td>881</td>
</tr>
<tr>
<td>file-filter-map</td>
<td>881</td>
</tr>
<tr>
<td>file-gid</td>
<td>886</td>
</tr>
<tr>
<td>file-info</td>
<td>746</td>
</tr>
<tr>
<td>file-info-device</td>
<td>747</td>
</tr>
<tr>
<td>file-info-directory</td>
<td>747</td>
</tr>
<tr>
<td>file-info-fifo</td>
<td>747</td>
</tr>
<tr>
<td>file-info-regular</td>
<td>747</td>
</tr>
<tr>
<td>file-info-socket</td>
<td>747</td>
</tr>
<tr>
<td>file-info-symlink</td>
<td>747</td>
</tr>
<tr>
<td>file-info-atime</td>
<td>747</td>
</tr>
<tr>
<td>file-info:bksize</td>
<td>747</td>
</tr>
<tr>
<td>file-info:blocks</td>
<td>747</td>
</tr>
<tr>
<td>file-info:ctime</td>
<td>747</td>
</tr>
<tr>
<td>file-info:device</td>
<td>747</td>
</tr>
<tr>
<td>file-info:gid</td>
<td>747</td>
</tr>
<tr>
<td>file-info:mode</td>
<td>747</td>
</tr>
<tr>
<td>file-info:mtime</td>
<td>747</td>
</tr>
<tr>
<td>file-info:nlinks</td>
<td>747</td>
</tr>
<tr>
<td>file-info:rdev</td>
<td>747</td>
</tr>
<tr>
<td>file-info:size</td>
<td>747</td>
</tr>
<tr>
<td>file-info:uid</td>
<td>747</td>
</tr>
<tr>
<td>file-info?</td>
<td>886</td>
</tr>
<tr>
<td>file-is-directory</td>
<td>301</td>
</tr>
<tr>
<td>file-is-executable</td>
<td>887</td>
</tr>
<tr>
<td>file-is-readable</td>
<td>887</td>
</tr>
<tr>
<td>file-is-regular</td>
<td>301</td>
</tr>
<tr>
<td>file-is-symlink</td>
<td>887</td>
</tr>
<tr>
<td>file-is-writable</td>
<td>887</td>
</tr>
<tr>
<td>file-mode</td>
<td>886</td>
</tr>
<tr>
<td>file-mtime</td>
<td>887</td>
</tr>
<tr>
<td>file-mtime&lt;=?</td>
<td>887</td>
</tr>
<tr>
<td>file-mtime&lt;&gt;?</td>
<td>887</td>
</tr>
<tr>
<td>file-mtime=?</td>
<td>887</td>
</tr>
<tr>
<td>file-mtime=&gt;?</td>
<td>887</td>
</tr>
<tr>
<td>file-nlink</td>
<td>886</td>
</tr>
<tr>
<td>file-perm</td>
<td>886</td>
</tr>
<tr>
<td>file-red</td>
<td>886</td>
</tr>
<tr>
<td>file-size</td>
<td>886</td>
</tr>
<tr>
<td>file-space</td>
<td>748</td>
</tr>
<tr>
<td>file-type</td>
<td>886</td>
</tr>
<tr>
<td>files</td>
<td>691</td>
</tr>
<tr>
<td>filter</td>
<td>149, 408</td>
</tr>
<tr>
<td>filter!</td>
<td>149</td>
</tr>
<tr>
<td>filter$</td>
<td>220</td>
</tr>
<tr>
<td>filter-map</td>
<td>149</td>
</tr>
<tr>
<td>filter-to</td>
<td>408</td>
</tr>
<tr>
<td>find</td>
<td>150, 408</td>
</tr>
<tr>
<td>find$</td>
<td>221</td>
</tr>
<tr>
<td>find-file-in-paths</td>
<td>886</td>
</tr>
<tr>
<td>find-gauche-package-description</td>
<td>479</td>
</tr>
<tr>
<td>find-in-queue</td>
<td>840</td>
</tr>
<tr>
<td>find-index</td>
<td>512</td>
</tr>
<tr>
<td>find-max</td>
<td>408</td>
</tr>
<tr>
<td>find-min</td>
<td>408</td>
</tr>
<tr>
<td>find-minmax</td>
<td>408</td>
</tr>
<tr>
<td>find-module</td>
<td>84</td>
</tr>
</tbody>
</table>
G

gamma ........................................... 133
gap-buffer->generator .................... 998
gap-buffer->string ......................... 998
gap-buffer-capacity ....................... 998
gap-buffer-clear ........................... 999
gap-buffer-contains ....................... 999
gap-buffer-content-length ............... 999
gap-buffer-copy .............................. 998
gap-buffer-delete! ......................... 999
gap-buffer-edit! ............................. 999
gap-buffer-gap-at? ......................... 999
gap-buffer-insert! ......................... 999
gap-buffer-looking-at? .................... 999
gap-buffer-move! ............................ 999
gap-buffer-pos ............................... 998
gap-buffer-pos-at-end? .................... 998
gap-buffer-ref ............................... 999
gap-buffer-replace! ......................... 999
gap-buffer-set! .............................. 999
gappend ....................................... 440
gaus-architecture ......................... 296
gaus-architecture-directory .......... 296
gaus-character-encoding ................. 164
gaus-config .................................. 413
gaus-library-directory ................. 296
gaus-package-description-paths ........ 479
gaus-site-architecture-directory ...... 296
gaus-site-library-directory .......... 296
gaus-thread-type ......................... 530
gaus-version ................................. 296
gbuffer-filter ............................... 443
gc ............................................. 323
gc-stat ...................................... 323
gchoicede .............................. 793
gcombine ..................................... 442
gcompose-left ................................ 792
gcompose-right .............................. 792
gconcatenate ................................ 440
gcons* ....................................... 440
gdbm-close ................................. 877
gdbm-closed? ............................... 877
gdbm-delete ................................. 877
gdbm-errno ................................ 878
gdbm-exists? ............................... 877
gdbm-fetch ................................. 577
gdbm-firstkey .............................. 877
gdbm-nextkey ............................... 877
gdbm-open .................................. 877
gdbm-reorganize ......................... 877
gdbm-setopt ............................... 878
gdbm-store ................................. 877
gdbm-sterror ............................... 878
gdbm-sync .................................. 877
gdbm-version ............................... 878
g delete .............................. 442
gdelete-neighbor-dups .................. 442
gdrop ....................................... 443
gdrop-while .............................. 444
generate .................................. 437
generation-either ....................... 770
generation->maybe ....................... 770
generator ................................... 440
generator->bytevector ................... 446
generator->bytevector! .................. 446
generator->seq ............................ 816
generator->idseq .......................... 626
generator->list ........................... 445
generator->laeq .......................... 242
generator->laeq/position .............. 455
generator->reverse-list ................. 445
generator->stream ......................... 793, 1036
generator->string ......................... 445
generator->uvector ....................... 446
generator->uvector! ...................... 446
generator->vector ......................... 445
generator->vector! ....................... 446
generator-any ............................. 447
generator-count ............................ 447
generator-every ............................ 447
generator-find ............................. 232
generator-fold ............................. 231
generator-fold-right ..................... 231
generator-for-each ....................... 232
generator-map ................................ 232
generator-map->list ...................... 445
generator-unfold ........................... 447
genym ........................................ 156
genumerate .................................... 793
g-environment-variable .................... 727
g-environment-variables .................. 727
get-f16 ..................................... 809
get-f16be .................................. 810
get-f16le ................................... 810
get-f32 ..................................... 809
get-f32be .................................. 810
get-f32le .................................. 810
get-f64 ..................................... 810
get-f64be .................................. 810
get-f64le .................................. 810
get-keyword* ............................... 157
get-keyword*# ............................... 158
get-optional ............................... 223
get-output-bytevector ................. 577
get-output-string .......................... 269
get-output-uvector ....................... 577
get-remaining-input-generator ........ 578
get-remaining-input-list ............... 577
get-remaining-input-string .......... 268
get-s16 .................................... 809
get-s16be .................................. 810
get-s16le .................................. 810
get-s22 ................................. 809
get-s32be .................................. 810
get-s32le .................................. 810
get-s64 .................................... 809
get-s64be .................................. 810
get-s64le .................................. 810
get-s8 ..................................... 809
get-signal-handler ....................... 311
get-signal-handler-mask ................ 311
get-signal-handles ...................... 311
get-signal-pending-limit .............. 311
get-sint ................................... 810
get-u16 .................................... 809
get-u16be .................................. 810
get-u16le .................................. 810
Appendix B: Function and Syntax Index 1087

get-u32 ............................ 809
get-u32be ................................ 810
get-u32le ................................ 810
get-u64 .................. 809
get-u64be ................................ 810
get-u64le ......................... 810
get-u8 .................................. 809
get-uint .................................. 810
getch .................................. 987
getter-with-setter ............... 53
gettext ................................ 1001
gfilter ................................ 442
gfilter-map ................................ 442
gflatten ................................ 441
ggroup ................................ 444
gindex .................................. 445
giota .................................. 436
giterate ................................ 439
giteratel ................................ 439
glet* .................................. 446
glet1 .................................. 447
glob .................................. 297
glob-fold .................................. 298
global-variable-bound? ....... 85
global-variable-ref ............. 85
gmap ................................ 441
gmap-accum ................................ 442
gmerge ................................ 441
goto .................................. 293
grange ................................ 437
greatest-fixnum .................. 133
gremove ................................ 442
group-by ................................ 806
group-collection .................... 409
group-contiguous-sequence ............ 514
group-info ................................ 749
group-info:gid ....................... 749
group-info:name ..................... 749
group-info ................................ 749
group-sequence ....................... 514
grxmatch ....... 444

gselect ................................ 445
gslices .................................. 446
gstate-filter 442
gtake .................................. 443
gtake* .................................. 443
gtake-while ................................ 444
guard .................................. 249
gunfold ................................ 439
gzip-decode-string .............. 959
gzip-encode-string ............. 959

H

has-setter? ......................... 53
has-windows-console? ............ 521
hash ................................ 115, 621, 723
hash-bound ......................... 116
hash-by-identity .................... 621, 723
hash-salt ................................ 116
hash-table ................................ 208, 618
hash-table-alist ......................... 209
hash-table-clear! ..................... 210
hash-table-comparator ............. 207
hash-table-compare-as-sets ....... 211
hash-table-contains? ............... 209
hash-table-copy ....................... 208
hash-table-count ..................... 620
hash-table-delete! ................... 210, 619
hash-table-delete!-r7 ............... 210
hash-table-difference! .......... 212
hash-table-empty-copy .......... 208
hash-table-equivalence-function ... 621, 722
hash-table-exists? ................. 209
hash-table-find ...................... 211, 619
hash-table-fold ...................... 211, 620
hash-table-for-each .................. 211, 620
hash-table-from-pairs ............. 207
hash-table-get ...................... 209
hash-table-hash-function ........ 621, 722
hash-table-internal! ............... 619
hash-table-internal!-r7 ............. 209
hash-table-intersection! ........... 212
hash-table-keys ..................... 211
hash-table-map ..................... 211, 620
hash-table-map! ..................... 620
hash-table-map->list ................ 620
hash-table-merge! .................. 621, 723
hash-table-mutable? ............... 207
hash-table-num-entries ............ 207
hash-table-pop! ..................... 210, 619
hash-table-pop!-r7 ............... 210
hash-table-prune! ................... 621
hash-table-push! ..................... 210
hash-table-put! ..................... 209
hash-table-r7 ....................... 208
hash-table-ref ...................... 209, 722
hash-table-ref/default ........... 209, 722
hash-table-set! ..................... 209, 722
hash-table-size ..................... 207, 722
hash-table-type .................... 207
hash-table-unfold .................. 208, 619
hash-table-union! .................. 212
hash-table-update! ............... 210, 619, 722
hash-table-update!-r7 .................. 211
hash-table-update!/default ...... 211, 722
hash-table-values .................. 211
hash-table-walk .................... 722
hash-table-xor! ..................... 212
hash-table=? ......................... 212
hash-table? ......................... 207
hashmap ................................ 659
hashmap->alist ...................... 662
hashmap-adjoin ...................... 661
hashmap-adjoin! ..................... 661
hashmap-any? ......................... 661
hashmap-contains? .................... 660
hashmap-copy ......................... 662
hashmap-count ......................... 662
hashmap-delete ....................... 661
hashmap-delete! ....................... 661
hashmap-delete-all .................. 661
hashmap-delete-all! .................. 661
hashmap-difference .................. 663
hashmap-difference! ............... 663
hashmap-disjoint? .................. 660
hashmap-empty? ....................... 660
hashmap-entries ..................... 662
hashmap-every? ....................... 661
Appendix B: Function and Syntax Index 1088

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashmap-filter</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-filter!</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-find</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-fold</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-for-each</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-intern</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-intern!</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-intersection</td>
<td>663</td>
</tr>
<tr>
<td>hashmap-intersection!</td>
<td>663</td>
</tr>
<tr>
<td>hashmap-key-comparator</td>
<td>660</td>
</tr>
<tr>
<td>hashmap-keys</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-map</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-map-&gt;list</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-partition</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-partition!</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-pop</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-pop!</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-ref</td>
<td>660</td>
</tr>
<tr>
<td>hashmap-ref/default</td>
<td>660</td>
</tr>
<tr>
<td>hashmap-remove</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-remove!</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-replace</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-replace!</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-search</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-search!</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-set</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-set!</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-size</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-unfold</td>
<td>659</td>
</tr>
<tr>
<td>hashmap-union</td>
<td>663</td>
</tr>
<tr>
<td>hashmap-union!</td>
<td>663</td>
</tr>
<tr>
<td>hashmap-update</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-update!</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-update/default</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-update/default</td>
<td>661</td>
</tr>
<tr>
<td>hashmap-values</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-xor</td>
<td>663</td>
</tr>
<tr>
<td>hashmap-xor!</td>
<td>663</td>
</tr>
<tr>
<td>hashmap-&lt;&lt;?</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-&lt;&lt;?</td>
<td>662</td>
</tr>
<tr>
<td>hashmap?&gt;&gt;?</td>
<td>662</td>
</tr>
<tr>
<td>hashmap?&gt;&gt;?</td>
<td>662</td>
</tr>
<tr>
<td>hashmap-?&gt;</td>
<td>662</td>
</tr>
<tr>
<td>hex-string-&gt;bytevector</td>
<td>777</td>
</tr>
<tr>
<td>hide-cursor</td>
<td>987</td>
</tr>
<tr>
<td>hmac-digest</td>
<td>930</td>
</tr>
<tr>
<td>hmac-digest-string</td>
<td>930</td>
</tr>
<tr>
<td>hmac-final!</td>
<td>929</td>
</tr>
<tr>
<td>hmac-message</td>
<td>929</td>
</tr>
<tr>
<td>hmac-message-to</td>
<td>929</td>
</tr>
<tr>
<td>hmac-to</td>
<td>928</td>
</tr>
<tr>
<td>hmac-update!</td>
<td>929</td>
</tr>
<tr>
<td>hmac-verify</td>
<td>929</td>
</tr>
<tr>
<td>home-directory</td>
<td>881</td>
</tr>
<tr>
<td>hook-&gt;list</td>
<td>448</td>
</tr>
<tr>
<td>hook-add!</td>
<td>751</td>
</tr>
<tr>
<td>hook-delete!</td>
<td>751</td>
</tr>
<tr>
<td>hook-empty?</td>
<td>448</td>
</tr>
<tr>
<td>hook-reset!</td>
<td>751</td>
</tr>
<tr>
<td>hook-run</td>
<td>751</td>
</tr>
<tr>
<td>hook?</td>
<td>448</td>
</tr>
<tr>
<td>html-doctype</td>
<td>1002</td>
</tr>
<tr>
<td>html-escape</td>
<td>1002</td>
</tr>
<tr>
<td>html-escape-string</td>
<td>1002</td>
</tr>
<tr>
<td>Function and Syntax</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
</tr>
<tr>
<td>html:strong</td>
<td>1002</td>
</tr>
<tr>
<td>html:style</td>
<td>1002</td>
</tr>
<tr>
<td>html:sub</td>
<td>1002</td>
</tr>
<tr>
<td>html:sup</td>
<td>1002</td>
</tr>
<tr>
<td>html:table</td>
<td>1002</td>
</tr>
<tr>
<td>html:tbody</td>
<td>1002</td>
</tr>
<tr>
<td>html:td</td>
<td>1002</td>
</tr>
<tr>
<td>html:textarea</td>
<td>1002</td>
</tr>
<tr>
<td>html:tfoot</td>
<td>1002</td>
</tr>
<tr>
<td>html:th</td>
<td>1002</td>
</tr>
<tr>
<td>html:thead</td>
<td>1002</td>
</tr>
<tr>
<td>html:title</td>
<td>1002</td>
</tr>
<tr>
<td>html:tr</td>
<td>1002</td>
</tr>
<tr>
<td>html:tt</td>
<td>1002</td>
</tr>
<tr>
<td>html:ul</td>
<td>1002</td>
</tr>
<tr>
<td>html:var</td>
<td>1002</td>
</tr>
<tr>
<td>http-compose-form-data</td>
<td>934</td>
</tr>
<tr>
<td>http-compose-query</td>
<td>934</td>
</tr>
<tr>
<td>http-default-redirect-handler</td>
<td>933</td>
</tr>
<tr>
<td>http-delete</td>
<td>930</td>
</tr>
<tr>
<td>http-get</td>
<td>930</td>
</tr>
<tr>
<td>http-head</td>
<td>930</td>
</tr>
<tr>
<td>http-post</td>
<td>930</td>
</tr>
<tr>
<td>http-proxy</td>
<td>933</td>
</tr>
<tr>
<td>http-put</td>
<td>930</td>
</tr>
<tr>
<td>http-secure-connection-available?</td>
<td>935</td>
</tr>
<tr>
<td>http-status-code-&gt;description</td>
<td>935</td>
</tr>
<tr>
<td>html:title</td>
<td>933</td>
</tr>
</tbody>
</table>

I

<table>
<thead>
<tr>
<th>Function and Syntax</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>i/o-decoding-error?</td>
<td>763</td>
</tr>
<tr>
<td>i/o-encoding-error-char</td>
<td>763</td>
</tr>
<tr>
<td>i/o-encoding-error?</td>
<td>763</td>
</tr>
<tr>
<td>i/o-invalid-position-error?</td>
<td>672</td>
</tr>
<tr>
<td>iapply</td>
<td>621</td>
</tr>
<tr>
<td>iassoc</td>
<td>621</td>
</tr>
<tr>
<td>iassq</td>
<td>621</td>
</tr>
<tr>
<td>iassv</td>
<td>621</td>
</tr>
<tr>
<td>icaaaar</td>
<td>621</td>
</tr>
<tr>
<td>icaaadr</td>
<td>621</td>
</tr>
<tr>
<td>icaaar</td>
<td>621</td>
</tr>
<tr>
<td>icaad</td>
<td>621</td>
</tr>
<tr>
<td>icaaddr</td>
<td>621</td>
</tr>
<tr>
<td>icaadr</td>
<td>621</td>
</tr>
<tr>
<td>icaaar</td>
<td>621</td>
</tr>
<tr>
<td>icadadr</td>
<td>621</td>
</tr>
<tr>
<td>icadar</td>
<td>621</td>
</tr>
<tr>
<td>icadardr</td>
<td>621</td>
</tr>
<tr>
<td>icaddr</td>
<td>621</td>
</tr>
<tr>
<td>icadddr</td>
<td>621</td>
</tr>
<tr>
<td>icadr</td>
<td>621</td>
</tr>
<tr>
<td>icdr</td>
<td>621</td>
</tr>
<tr>
<td>icddadr</td>
<td>621</td>
</tr>
<tr>
<td>icddar</td>
<td>621</td>
</tr>
<tr>
<td>icddd</td>
<td>621</td>
</tr>
<tr>
<td>icdddar</td>
<td>621</td>
</tr>
<tr>
<td>icdddar</td>
<td>621</td>
</tr>
<tr>
<td>icdaddr</td>
<td>621</td>
</tr>
<tr>
<td>icdd</td>
<td>621</td>
</tr>
<tr>
<td>icddr</td>
<td>621</td>
</tr>
<tr>
<td>icdr</td>
<td>621</td>
</tr>
<tr>
<td>icdar</td>
<td>621</td>
</tr>
<tr>
<td>icddaaar</td>
<td>621</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>list-queue-add-front!</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-append</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-append!</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-back</td>
<td>637</td>
</tr>
<tr>
<td>list-queue-concatenate</td>
<td>639</td>
</tr>
<tr>
<td>list-queue-copy</td>
<td>637</td>
</tr>
<tr>
<td>list-queue-empty?</td>
<td>637</td>
</tr>
<tr>
<td>list-queue-fist-last</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-for-each</td>
<td>639</td>
</tr>
<tr>
<td>list-queue-front</td>
<td>637</td>
</tr>
<tr>
<td>list-queue-list</td>
<td>637</td>
</tr>
<tr>
<td>list-queue-map</td>
<td>639</td>
</tr>
<tr>
<td>list-queue-map!</td>
<td>639</td>
</tr>
<tr>
<td>list-queue-remove-all!</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-remove-back!</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-remove-front!</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-set-list!</td>
<td>638</td>
</tr>
<tr>
<td>list-queue-unfold</td>
<td>637</td>
</tr>
<tr>
<td>list-queue-unfold-right</td>
<td>637</td>
</tr>
<tr>
<td>list-queue?</td>
<td>637</td>
</tr>
<tr>
<td>list-ref</td>
<td>146</td>
</tr>
<tr>
<td>list-ref/update</td>
<td>623</td>
</tr>
<tr>
<td>list-set</td>
<td>623</td>
</tr>
<tr>
<td>list-set!</td>
<td>146</td>
</tr>
<tr>
<td>list-sort</td>
<td>603</td>
</tr>
<tr>
<td>list-sort!</td>
<td>603</td>
</tr>
<tr>
<td>list-sort?</td>
<td>603</td>
</tr>
<tr>
<td>list-stable-sort</td>
<td>603</td>
</tr>
<tr>
<td>list-stable-sort!</td>
<td>603</td>
</tr>
<tr>
<td>list-tabulate</td>
<td>593</td>
</tr>
<tr>
<td>list-tail</td>
<td>146</td>
</tr>
<tr>
<td>list-truth-&gt;either</td>
<td>770</td>
</tr>
<tr>
<td>list-truth-&gt;maybe</td>
<td>769</td>
</tr>
<tr>
<td>list-values</td>
<td>785</td>
</tr>
<tr>
<td>list/mv</td>
<td>785</td>
</tr>
<tr>
<td>list=</td>
<td>593</td>
</tr>
<tr>
<td>list?</td>
<td>142</td>
</tr>
<tr>
<td>listener-read-handler</td>
<td>457</td>
</tr>
<tr>
<td>listener-show-prompt</td>
<td>457</td>
</tr>
<tr>
<td>lists-of</td>
<td>845</td>
</tr>
<tr>
<td>literate</td>
<td>451</td>
</tr>
<tr>
<td>list*</td>
<td>243</td>
</tr>
<tr>
<td>lmap</td>
<td>452</td>
</tr>
<tr>
<td>lmap-accum</td>
<td>452</td>
</tr>
<tr>
<td>load</td>
<td>286</td>
</tr>
<tr>
<td>load-bundle!</td>
<td>590</td>
</tr>
<tr>
<td>load-from-port</td>
<td>708</td>
</tr>
<tr>
<td>localized-template</td>
<td>287</td>
</tr>
<tr>
<td>log</td>
<td>708</td>
</tr>
<tr>
<td>log-default-drain</td>
<td>131</td>
</tr>
<tr>
<td>log-format</td>
<td>460</td>
</tr>
<tr>
<td>log-open</td>
<td>460</td>
</tr>
<tr>
<td>log2-binary-factors</td>
<td>719</td>
</tr>
<tr>
<td>logand</td>
<td>137</td>
</tr>
<tr>
<td>logand=?</td>
<td>395</td>
</tr>
<tr>
<td>logbit?</td>
<td>137</td>
</tr>
<tr>
<td>logcount</td>
<td>138</td>
</tr>
<tr>
<td>logior</td>
<td>137</td>
</tr>
<tr>
<td>logior=?</td>
<td>395</td>
</tr>
<tr>
<td>lognot</td>
<td>137</td>
</tr>
<tr>
<td>logset+clear</td>
<td>137</td>
</tr>
<tr>
<td>logtest</td>
<td>137</td>
</tr>
<tr>
<td>logxor</td>
<td>137</td>
</tr>
<tr>
<td>logxor=?</td>
<td>395</td>
</tr>
<tr>
<td>loop</td>
<td>392</td>
</tr>
<tr>
<td>loop-procedure</td>
<td>807</td>
</tr>
<tr>
<td>lrange</td>
<td>243</td>
</tr>
<tr>
<td>lrxmatch</td>
<td>454</td>
</tr>
<tr>
<td>lseq-&gt;generator</td>
<td>634</td>
</tr>
<tr>
<td>lseq-&gt;list</td>
<td>452</td>
</tr>
<tr>
<td>lseq-anv</td>
<td>635</td>
</tr>
<tr>
<td>lseq-append</td>
<td>635</td>
</tr>
<tr>
<td>lseq-car</td>
<td>634</td>
</tr>
<tr>
<td>lseq-cdr</td>
<td>634</td>
</tr>
<tr>
<td>lseq-drop</td>
<td>634</td>
</tr>
<tr>
<td>lseq-drop-while</td>
<td>635</td>
</tr>
<tr>
<td>lseq-every</td>
<td>635</td>
</tr>
<tr>
<td>lseq-filter</td>
<td>635</td>
</tr>
<tr>
<td>lseq-find</td>
<td>635</td>
</tr>
<tr>
<td>lseq-find+tail</td>
<td>634</td>
</tr>
<tr>
<td>lseq-first</td>
<td>634</td>
</tr>
<tr>
<td>lseq-for-each</td>
<td>635</td>
</tr>
<tr>
<td>lseq-index</td>
<td>635</td>
</tr>
<tr>
<td>lseq-length</td>
<td>634</td>
</tr>
<tr>
<td>lseq-map</td>
<td>635</td>
</tr>
<tr>
<td>lseq-member</td>
<td>635</td>
</tr>
<tr>
<td>lseq-memq</td>
<td>635</td>
</tr>
<tr>
<td>lseq-memv</td>
<td>635</td>
</tr>
<tr>
<td>lseq-position</td>
<td>455</td>
</tr>
<tr>
<td>lseq-realize</td>
<td>634</td>
</tr>
<tr>
<td>lseq-remove</td>
<td>635</td>
</tr>
<tr>
<td>lseq-rest</td>
<td>634</td>
</tr>
<tr>
<td>lseq-take</td>
<td>634</td>
</tr>
<tr>
<td>lseq-take-while</td>
<td>635</td>
</tr>
<tr>
<td>lseq=?</td>
<td>634</td>
</tr>
<tr>
<td>lseq?</td>
<td>634</td>
</tr>
<tr>
<td>lset-adjoin</td>
<td>597</td>
</tr>
<tr>
<td>lset-diff+intersection</td>
<td>597</td>
</tr>
<tr>
<td>lset-diff+intersection!</td>
<td>597</td>
</tr>
<tr>
<td>lset-difference</td>
<td>597</td>
</tr>
<tr>
<td>lset-difference!</td>
<td>597</td>
</tr>
<tr>
<td>lset-intersection</td>
<td>597</td>
</tr>
<tr>
<td>lset-intersection!</td>
<td>597</td>
</tr>
<tr>
<td>lset-union</td>
<td>597</td>
</tr>
<tr>
<td>lset-union!</td>
<td>597</td>
</tr>
<tr>
<td>lset-xor</td>
<td>597</td>
</tr>
<tr>
<td>lset-xor!</td>
<td>597</td>
</tr>
<tr>
<td>lset&lt;=</td>
<td>596</td>
</tr>
<tr>
<td>lset=</td>
<td>597</td>
</tr>
<tr>
<td>lslises</td>
<td>454</td>
</tr>
<tr>
<td>lstate-filter</td>
<td>453</td>
</tr>
<tr>
<td>ltake</td>
<td>454</td>
</tr>
<tr>
<td>ltake-while</td>
<td>454</td>
</tr>
<tr>
<td>lumfold</td>
<td>451</td>
</tr>
</tbody>
</table>

### M

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine-name</td>
<td>730</td>
</tr>
<tr>
<td>macroexpand</td>
<td>102</td>
</tr>
<tr>
<td>macroexpand-1</td>
<td>102</td>
</tr>
<tr>
<td>macroexpand-all</td>
<td>103</td>
</tr>
<tr>
<td>magnitude</td>
<td>134</td>
</tr>
<tr>
<td>make</td>
<td>345-357</td>
</tr>
<tr>
<td>make-@vector-generator</td>
<td>562</td>
</tr>
<tr>
<td>make-accumulator</td>
<td>632</td>
</tr>
<tr>
<td>make-array</td>
<td>367</td>
</tr>
<tr>
<td>make-barrier</td>
<td>543</td>
</tr>
<tr>
<td>make-bimap</td>
<td>430</td>
</tr>
<tr>
<td>make-binary-heap</td>
<td>832</td>
</tr>
<tr>
<td>Function Name</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>make-bitvector</td>
<td>204</td>
</tr>
<tr>
<td>make-bitvector-accumulator</td>
<td>379</td>
</tr>
<tr>
<td>make-bitvector/bool-generator</td>
<td>379</td>
</tr>
<tr>
<td>make-bitvector/int-generator</td>
<td>379</td>
</tr>
<tr>
<td>make-bitwise-generator</td>
<td>668</td>
</tr>
<tr>
<td>make-blob</td>
<td>723</td>
</tr>
<tr>
<td>make-byte-string</td>
<td>172</td>
</tr>
<tr>
<td>make-bytestring</td>
<td>776</td>
</tr>
<tr>
<td>make-bytestring!</td>
<td>776</td>
</tr>
<tr>
<td>make-bytestring-generator</td>
<td>777</td>
</tr>
<tr>
<td>make-bytevector</td>
<td>569</td>
</tr>
<tr>
<td>make-bytevector-comparator</td>
<td>733</td>
</tr>
<tr>
<td>make-c12vector</td>
<td>260</td>
</tr>
<tr>
<td>make-c32vector</td>
<td>206</td>
</tr>
<tr>
<td>make-c64vector</td>
<td>200</td>
</tr>
<tr>
<td>make-car-comparator</td>
<td>733</td>
</tr>
<tr>
<td>make-cdr-comparator</td>
<td>733</td>
</tr>
<tr>
<td>make-client-socket</td>
<td>467, 728</td>
</tr>
<tr>
<td>make-codex</td>
<td>763</td>
</tr>
<tr>
<td>make-comparator</td>
<td>117, 732</td>
</tr>
<tr>
<td>make-comparator/compare</td>
<td>117</td>
</tr>
<tr>
<td>make-comparison&lt;</td>
<td>735</td>
</tr>
<tr>
<td>make-comparison&lt;=</td>
<td>735</td>
</tr>
<tr>
<td>make-comparison/=</td>
<td>735</td>
</tr>
<tr>
<td>make-comparison&gt;</td>
<td>735</td>
</tr>
<tr>
<td>make-compound</td>
<td>794</td>
</tr>
<tr>
<td>make-compound-condition</td>
<td>257</td>
</tr>
<tr>
<td>make-condition</td>
<td>257</td>
</tr>
<tr>
<td>make-condition-type</td>
<td>257</td>
</tr>
<tr>
<td>make-condition-variable</td>
<td>539</td>
</tr>
<tr>
<td>make-continuation-mark-key</td>
<td>230</td>
</tr>
<tr>
<td>make-continuation-prompt-tag</td>
<td>230</td>
</tr>
<tr>
<td>make-continuation-violation</td>
<td>256</td>
</tr>
<tr>
<td>make-coroutine-generator</td>
<td>440</td>
</tr>
<tr>
<td>make-csv-header-parser</td>
<td>991</td>
</tr>
<tr>
<td>make-csv-reader</td>
<td>894</td>
</tr>
<tr>
<td>make-csv-record-parser</td>
<td>991</td>
</tr>
<tr>
<td>make-custom-binary-input-port</td>
<td>760</td>
</tr>
<tr>
<td>make-custom-binary-input/output-port</td>
<td>761</td>
</tr>
<tr>
<td>make-custom-binary-output-port</td>
<td>760</td>
</tr>
<tr>
<td>make-custom-textual-input-port</td>
<td>760</td>
</tr>
<tr>
<td>make-custom-textual-output-port</td>
<td>760</td>
</tr>
<tr>
<td>make-date</td>
<td>704</td>
</tr>
<tr>
<td>make-debug-comparator</td>
<td>734</td>
</tr>
<tr>
<td>make-default-comparator</td>
<td>121</td>
</tr>
<tr>
<td>make-default-console</td>
<td>986</td>
</tr>
<tr>
<td>make-directory-files-generator</td>
<td>747</td>
</tr>
<tr>
<td>make-edn-object</td>
<td>996</td>
</tr>
<tr>
<td>make-empty-attlist</td>
<td>962</td>
</tr>
<tr>
<td>make-enum-comparator</td>
<td>781</td>
</tr>
<tr>
<td>make-enum-type</td>
<td>780</td>
</tr>
<tr>
<td>make-enumeration</td>
<td>784</td>
</tr>
<tr>
<td>make-ephemeron</td>
<td>640</td>
</tr>
<tr>
<td>make-eq-comparator</td>
<td>121</td>
</tr>
<tr>
<td>make-eqv-comparator</td>
<td>121</td>
</tr>
<tr>
<td>make-f16array</td>
<td>367</td>
</tr>
<tr>
<td>make-f16vector</td>
<td>200</td>
</tr>
<tr>
<td>make-f32array</td>
<td>367</td>
</tr>
<tr>
<td>make-f32vector</td>
<td>290</td>
</tr>
<tr>
<td>make-f64array</td>
<td>367</td>
</tr>
<tr>
<td>make-range-generator</td>
<td>440</td>
</tr>
<tr>
<td>make-range-set</td>
<td>788</td>
</tr>
<tr>
<td>make-rbtree</td>
<td>364</td>
</tr>
<tr>
<td>make-record-type</td>
<td>1030</td>
</tr>
<tr>
<td>make-rectangular</td>
<td>134</td>
</tr>
</tbody>
</table>
Appendix B: Function and Syntax Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>module-reload-rules</td>
<td>508</td>
</tr>
<tr>
<td>module-table</td>
<td>85</td>
</tr>
<tr>
<td>module?</td>
<td>84</td>
</tr>
<tr>
<td>modulo</td>
<td>127</td>
</tr>
<tr>
<td>monotonic-time</td>
<td>749</td>
</tr>
<tr>
<td>move-cursor-to</td>
<td>987</td>
</tr>
<tr>
<td>move-file</td>
<td>889</td>
</tr>
<tr>
<td>mt-random-fill-f32vector!</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-fill-f64vector!</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-fill-u32vector!</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-get-seed</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-get-state</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-integer</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-real</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-real0</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-set-seed!</td>
<td>894</td>
</tr>
<tr>
<td>mt-random-set-state!</td>
<td>894</td>
</tr>
<tr>
<td>mtqueue-max-length</td>
<td>838</td>
</tr>
<tr>
<td>mtqueue-num-waiting-readers</td>
<td>838</td>
</tr>
<tr>
<td>mtqueue-room</td>
<td>838</td>
</tr>
<tr>
<td>mutex?</td>
<td>537</td>
</tr>
<tr>
<td>mutex-lock!</td>
<td>537</td>
</tr>
<tr>
<td>mutex-locker</td>
<td>537</td>
</tr>
<tr>
<td>mutex-name</td>
<td>537</td>
</tr>
<tr>
<td>mutex-specific</td>
<td>537</td>
</tr>
<tr>
<td>mutex-specific-set!</td>
<td>537</td>
</tr>
<tr>
<td>mutex-state</td>
<td>537</td>
</tr>
<tr>
<td>mutex-unlock!</td>
<td>538</td>
</tr>
<tr>
<td>mutex-unlocker</td>
<td>538</td>
</tr>
<tr>
<td>mutex?</td>
<td>537</td>
</tr>
<tr>
<td>node-equal?</td>
<td>971</td>
</tr>
<tr>
<td>node-join</td>
<td>973</td>
</tr>
<tr>
<td>node-or</td>
<td>973</td>
</tr>
<tr>
<td>node-pos</td>
<td>971</td>
</tr>
<tr>
<td>node-reduce</td>
<td>973</td>
</tr>
<tr>
<td>node-reverse</td>
<td>972</td>
</tr>
<tr>
<td>node-self</td>
<td>973</td>
</tr>
<tr>
<td>node-trace</td>
<td>972</td>
</tr>
<tr>
<td>node-set?</td>
<td>970</td>
</tr>
<tr>
<td>null?</td>
<td>139, 394, 716</td>
</tr>
<tr>
<td>not-ipair?</td>
<td>621</td>
</tr>
<tr>
<td>not-pair?</td>
<td>593</td>
</tr>
<tr>
<td>nothing</td>
<td>766</td>
</tr>
<tr>
<td>nothing?</td>
<td>765</td>
</tr>
<tr>
<td>ntype-names?</td>
<td>970</td>
</tr>
<tr>
<td>ntype-names-id?</td>
<td>971</td>
</tr>
<tr>
<td>ntype?</td>
<td>971</td>
</tr>
<tr>
<td>null-device</td>
<td>886</td>
</tr>
<tr>
<td>null-environment</td>
<td>258</td>
</tr>
<tr>
<td>null-generator</td>
<td>436</td>
</tr>
<tr>
<td>null-list?</td>
<td>621</td>
</tr>
<tr>
<td>null-list?</td>
<td>141</td>
</tr>
<tr>
<td>null?</td>
<td>141</td>
</tr>
<tr>
<td>num-pairs</td>
<td>145</td>
</tr>
<tr>
<td>number-string</td>
<td>136</td>
</tr>
<tr>
<td>number-hash</td>
<td>115</td>
</tr>
<tr>
<td>number?</td>
<td>123</td>
</tr>
<tr>
<td>numerator</td>
<td>130</td>
</tr>
<tr>
<td>numeric</td>
<td>683</td>
</tr>
<tr>
<td>numeric-range</td>
<td>846</td>
</tr>
<tr>
<td>numeric/comma</td>
<td>683</td>
</tr>
<tr>
<td>numeric/fitted</td>
<td>683</td>
</tr>
<tr>
<td>numeric/si</td>
<td>683</td>
</tr>
</tbody>
</table>

N
naive-factorize                         | 896  |
nam?                                    | 124  |
native-endian                          | 139  |
native-endianness                       | 679  |
native-eol-style                        | 763  |
native-transcoder                       | 763  |
ndbm-clear-error                        | 879  |
ndbm-close                              | 878  |
ndbm-closed?                            | 878  |
ndbm-delete                             | 878  |
ndbm-error                              | 879  |
ndbm-fetch                              | 878  |
ndbm-firstkey                           | 879  |
ndbm-nextkey                            | 879  |
ndbm-open                               | 878  |
ndbm-store                              | 878  |
negative-zero?                          | 124  |
negative?                              | 124  |
nest                                    | 776  |
nest-reverse                            | 776  |
nested                                  | 717  |
ever                                   | 807  |
newline                                 | 279  |
next-method                             | 351  |
next-token                              | 1008 |
next-token-of                           | 1008 |
nggettext                               | 1001 |
nice                                    | 748  |
nil-uuid                                | 955  |
sinth                                   | 594  |
node-closure                            | 973  |
node-eq?                                | 971  |

O
object-apply                            | 225  |
object-compare                          | 113  |
object-equal?                           | 111, 115 |
object-hash                             | 114, 115 |
object-isomorphic?                      | 1020 |
odbm-close                              | 879  |
odbm-delete                             | 879  |
odbm-fetch                              | 879  |
odbm-firstkey                           | 879  |
odbm-init                               | 879  |
odbm-nextkey                            | 879  |
odbm-store                              | 879  |
odd?                                    | 124  |
of-type?                                | 106  |
on                                      | 805  |
on-left                                | 804  |
on-right                               | 804  |
open-binary-input-file                  | 589  |
open-binary-output-file                 | 589  |
open-broadcast-output-port             | 822  |
open-coding-aware-port                  | 270  |
open-deflating-port                     | 956  |
open-directory                          | 747  |
open-file                               | 745  |
open-inflating-port                     | 957  |
open-info-document                      | 1004 |
open-inlet-output-port                  | 820  |
open-input-output-port                  | 577  |
open-input-byte-generator               | 577  |
open-input-byte-list                    | 577  |
open-input-bytevector          576
open-input-char-generator      577
open-input-char-list          577
open-input-conversion-port    402
open-input-fd-port            267
open-input-file               264
open-input-process-port       498
open-input-string             268
open-input-uvector            576
open-output-input-port        820
open-output-accumulator       578
open-output-bytevector        576
open-output-char-accumulator  578
open-output-conversion-port   403
open-output-fd-port           267
open-output-file              264
open-output-process-port      499
open-output-string            268
open-output-uvector           576
open-tapping-input-port       822
opt**-labmda                  800
opt-labmda                    800
opt-substring                 178
option                        710
option-names                  710
option-optional-arg?          710
option-processor             710
option-required-arg?          710
option?                       710
or                            57, 394, 716
or-procedure                  806
os-name                       730
os-version                    730
output-port-open?             587
output-port?                  260

parse-cs-file                 1054
parse-cs-selector-string      1054
parse-edn                     994
parse-edn*                    995
parse-edn-string              995
parse-json                    937
parse-json*                   937
parse-json-string             937
parse-options                 484
parse-success?                910
parse-uuid                    955
partition                     409, 596
partition!                   596
partition?                   220
partition-to                  409
path->gauche-package-description 479
path->module-name             85
path-extension                885
path-sans-extension           885
path-swap-extension           885
peek-byte                     271
peek-char                     271
peek-next-char                1007
peek-u8                       271
peg-parse-port                907
peg-parse-string              907
peg-parser->generator         908
peg-run-parser                907
permutations                  1015
permutations*                 1015
permutations*+for-each        1015
permutations-for-each         1015
permutations-of               844
permute                       516
permute!                      517
permute-to                    517
pfind                         823
pid                            748
plumbing                      821
plumbing-get-port             820
plumbing-inlet-ports          820
plumbing-outlet-ports         820
plumbing?                     820
pmap                          823
pop!                          54
port->byte-generator          438
port->byte-lseq               438
port->char-generator          438
port->char-lseq               438
port->char-lseq/position      455
port->line-generator          438
port->list                    274
port->sexp-generator          438
port->sexp-lack               438
port->sexp-lseq               438
port->stream                  1036
port->string                  274
port->string-list             275
port->string-lseq             244
port->vector                  568
port->buffering               261
port->closed?                 260
port-current-line             262
port-fd-dup!                  267
port-file-number              262

P

pa$                          220
pack                         811
padded/                       684
padded/both                 685
padded/right                685
pager-program               1006
pair-attribute-get           155
pair-attribute-set!          155
pair-attributes             154
pair-fold                    595
pair-fold-right             595
pair-for-each               393, 595
pair?                         141
pairs-of                    844
pany                         823
parallel-execute            877
parameter-observer-add!      840
parameter-observer-delete!   480
parameter-post-observers     480
parameter-pre-observers      480
parameter?                  235
parameterization?           235
parameterize                235
parameterize/dynwind         235
parse-cookie-string          924
parse-cs                   1054
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>read-u64</td>
<td>808</td>
</tr>
<tr>
<td>read-u8</td>
<td>271, 808</td>
</tr>
<tr>
<td>read-uvector</td>
<td>808</td>
</tr>
<tr>
<td>read-uvector!</td>
<td>567</td>
</tr>
<tr>
<td>read-with-shared-structure</td>
<td>271</td>
</tr>
<tr>
<td>read/ss</td>
<td>271</td>
</tr>
<tr>
<td>reader-lexical-mode</td>
<td>272</td>
</tr>
<tr>
<td>real-&gt;rational</td>
<td>135</td>
</tr>
<tr>
<td>real-acos</td>
<td>132</td>
</tr>
<tr>
<td>real-acosh</td>
<td>133</td>
</tr>
<tr>
<td>real-asin</td>
<td>132</td>
</tr>
<tr>
<td>real-asinh</td>
<td>133</td>
</tr>
<tr>
<td>real-atan</td>
<td>133</td>
</tr>
<tr>
<td>real-atanh</td>
<td>133</td>
</tr>
<tr>
<td>real-cos</td>
<td>132</td>
</tr>
<tr>
<td>real-cosh</td>
<td>133</td>
</tr>
<tr>
<td>real-cospi</td>
<td>133</td>
</tr>
<tr>
<td>real-exp</td>
<td>132</td>
</tr>
<tr>
<td>real-expt</td>
<td>132</td>
</tr>
<tr>
<td>real-ln</td>
<td>132</td>
</tr>
<tr>
<td>real-part</td>
<td>134</td>
</tr>
<tr>
<td>real-path</td>
<td>748</td>
</tr>
<tr>
<td>real-sin</td>
<td>132</td>
</tr>
<tr>
<td>real-sinh</td>
<td>133</td>
</tr>
<tr>
<td>real-sinpi</td>
<td>133</td>
</tr>
<tr>
<td>real-sqrt</td>
<td>132</td>
</tr>
<tr>
<td>real-tan</td>
<td>132</td>
</tr>
<tr>
<td>real-tanh</td>
<td>133</td>
</tr>
<tr>
<td>real-tanpi</td>
<td>133</td>
</tr>
<tr>
<td>real-valued?</td>
<td>123</td>
</tr>
<tr>
<td>real?</td>
<td>123</td>
</tr>
<tr>
<td>reals$</td>
<td>842</td>
</tr>
<tr>
<td>reals-between$</td>
<td>842</td>
</tr>
<tr>
<td>reals-exponential$</td>
<td>843</td>
</tr>
<tr>
<td>reals-normal$</td>
<td>843</td>
</tr>
<tr>
<td>rec</td>
<td>62</td>
</tr>
<tr>
<td>receive</td>
<td>61</td>
</tr>
<tr>
<td>record-accessor</td>
<td>129</td>
</tr>
<tr>
<td>record-constructor</td>
<td>1030</td>
</tr>
<tr>
<td>record-modifier</td>
<td>1030</td>
</tr>
<tr>
<td>record-predicate</td>
<td>1030</td>
</tr>
<tr>
<td>record-rtd</td>
<td>505</td>
</tr>
<tr>
<td>record?</td>
<td>505</td>
</tr>
<tr>
<td>reduce</td>
<td>149</td>
</tr>
<tr>
<td>reduce$</td>
<td>220</td>
</tr>
<tr>
<td>reduce-right</td>
<td>149</td>
</tr>
<tr>
<td>reduce-right$</td>
<td>220</td>
</tr>
<tr>
<td>ref</td>
<td>209, 220, 347, 394, 510</td>
</tr>
<tr>
<td>reference-barrier</td>
<td>640</td>
</tr>
<tr>
<td>referencer</td>
<td>511</td>
</tr>
<tr>
<td>regexp</td>
<td>188, 649</td>
</tr>
<tr>
<td>regexp-&gt;sre</td>
<td>187, 649</td>
</tr>
<tr>
<td>regexp-&gt;string</td>
<td>187</td>
</tr>
<tr>
<td>regexp-ast</td>
<td>194</td>
</tr>
<tr>
<td>regexp-compile</td>
<td>193</td>
</tr>
<tr>
<td>regexp-extract</td>
<td>650</td>
</tr>
<tr>
<td>regexp-fold</td>
<td>650</td>
</tr>
<tr>
<td>regexp-match-&gt;list</td>
<td>652</td>
</tr>
<tr>
<td>regexp-match-count</td>
<td>652</td>
</tr>
<tr>
<td>regexp-match-submatch</td>
<td>652</td>
</tr>
<tr>
<td>regexp-match-submatch-end</td>
<td>652</td>
</tr>
<tr>
<td>regexp-match-submatch-start</td>
<td>652</td>
</tr>
<tr>
<td>regexp-match?</td>
<td>651</td>
</tr>
<tr>
<td>regexp-matches?</td>
<td>650</td>
</tr>
<tr>
<td>regexp-matched-groups</td>
<td>650</td>
</tr>
<tr>
<td>regexp-num-groups</td>
<td>187</td>
</tr>
<tr>
<td>regexp-optimize</td>
<td>193</td>
</tr>
<tr>
<td>regexp-parse</td>
<td>193</td>
</tr>
<tr>
<td>regexp-parse-sre</td>
<td>193</td>
</tr>
<tr>
<td>regexp-partition</td>
<td>651</td>
</tr>
<tr>
<td>regexp-quote</td>
<td>191</td>
</tr>
<tr>
<td>regexp-replace</td>
<td>190, 651</td>
</tr>
<tr>
<td>regexp-replace*</td>
<td>191</td>
</tr>
<tr>
<td>regexp-replace-all</td>
<td>190, 651</td>
</tr>
<tr>
<td>regexp-replace-all*</td>
<td>191</td>
</tr>
<tr>
<td>regexp-search</td>
<td>650</td>
</tr>
<tr>
<td>regexp-split</td>
<td>650</td>
</tr>
<tr>
<td>regexp-unparse</td>
<td>194</td>
</tr>
<tr>
<td>regexp?</td>
<td>187, 650</td>
</tr>
<tr>
<td>register-edn-object-handler!</td>
<td>996</td>
</tr>
<tr>
<td>regmatch</td>
<td>189</td>
</tr>
<tr>
<td>regular-string$</td>
<td>843</td>
</tr>
<tr>
<td>relate</td>
<td>1045</td>
</tr>
<tr>
<td>relate-point</td>
<td>1045</td>
</tr>
<tr>
<td>relation-accessor</td>
<td>1031</td>
</tr>
<tr>
<td>relation-coercer</td>
<td>1032</td>
</tr>
<tr>
<td>relation-column-getter</td>
<td>1032</td>
</tr>
<tr>
<td>relation-column-getters</td>
<td>1032</td>
</tr>
<tr>
<td>relation-column-name?</td>
<td>1032</td>
</tr>
<tr>
<td>relation-column-names</td>
<td>1031</td>
</tr>
<tr>
<td>relation-column-setter</td>
<td>1032</td>
</tr>
<tr>
<td>relation-column-setters</td>
<td>1032</td>
</tr>
<tr>
<td>relation-deletable?</td>
<td>1032</td>
</tr>
<tr>
<td>relation-delete!</td>
<td>1032</td>
</tr>
<tr>
<td>relation-fold</td>
<td>1032</td>
</tr>
<tr>
<td>relation-insert!</td>
<td>1032</td>
</tr>
<tr>
<td>relation-insertable?</td>
<td>1032</td>
</tr>
<tr>
<td>relation-modifier</td>
<td>1031</td>
</tr>
<tr>
<td>relation-ref</td>
<td>1032</td>
</tr>
<tr>
<td>relation-rows</td>
<td>1031</td>
</tr>
<tr>
<td>relation-set!</td>
<td>1032</td>
</tr>
<tr>
<td>relation?</td>
<td>1032</td>
</tr>
<tr>
<td>relative-path?</td>
<td>884</td>
</tr>
<tr>
<td>relnum-compare</td>
<td>572</td>
</tr>
<tr>
<td>reload</td>
<td>508</td>
</tr>
<tr>
<td>reload-modified-modules</td>
<td>508</td>
</tr>
<tr>
<td>reload-verbose</td>
<td>508</td>
</tr>
<tr>
<td>remainder</td>
<td>127</td>
</tr>
<tr>
<td>remove</td>
<td>150, 408</td>
</tr>
<tr>
<td>remove!</td>
<td>150</td>
</tr>
<tr>
<td>remove$</td>
<td>220</td>
</tr>
<tr>
<td>remove-directory*</td>
<td>883</td>
</tr>
<tr>
<td>remove-file</td>
<td>889</td>
</tr>
<tr>
<td>remove-files</td>
<td>889</td>
</tr>
<tr>
<td>remove-from-queue!</td>
<td>846</td>
</tr>
<tr>
<td>remove-hook!</td>
<td>448</td>
</tr>
<tr>
<td>remove-to</td>
<td>409</td>
</tr>
<tr>
<td>rename-file</td>
<td>746</td>
</tr>
<tr>
<td>report-error</td>
<td>252</td>
</tr>
<tr>
<td>report-time-results</td>
<td>548</td>
</tr>
<tr>
<td>require</td>
<td>288</td>
</tr>
<tr>
<td>require-extension</td>
<td>718</td>
</tr>
<tr>
<td>requires</td>
<td>691</td>
</tr>
<tr>
<td>reset</td>
<td>485</td>
</tr>
<tr>
<td>reset-character-attribute</td>
<td>988</td>
</tr>
<tr>
<td>reset-hook!</td>
<td>448</td>
</tr>
<tr>
<td>reset-primes</td>
<td>895</td>
</tr>
<tr>
<td>reset-terminal</td>
<td>987</td>
</tr>
</tbody>
</table>
resolve-path ........................................... 885
return ............................................... 393
return-failure ....................................... 909
return-failure!compound ............................... 909
return-failure!expect .................................. 909
return-failure!message ................................ 909
return-failure!unexpected .............................. 909
return-result ......................................... 909
reverse ................................................ 151
reverse .............................................. 151
reverse-bit-field ....................................... 719
reverse-bitvector->generator ........................... 438
reverse-bitvector->list/bool .......................... 378
reverse-bitvector->list/int ............................ 378
reverse-bitvector->vector/bool ........................ 378
reverse-list->0vector .................................. 564
reverse-list->bitvector ................................ 378
reverse-list->c128vector .............................. 564
reverse-list->c32vector ................................ 564
reverse-list->c64vector ................................ 564
reverse-list->f16vector ................................ 564
reverse-list->f32vector ................................ 564
reverse-list->f64vector ................................ 564
reverse-list->s16vector ................................ 564
reverse-list->s32vector ................................ 564
reverse-list->s64vector ................................ 564
reverse-list->s8vector ................................ 564
reverse-list->string ................................... 694
reverse-list->text ..................................... 628
reverse-list->u16vector ................................ 564
reverse-list->u32vector ................................ 564
reverse-list->u64vector ................................ 564
reverse-list->u8vector ................................ 564
reverse-list-vector .................................... 196
reverse-list-accumulator ................................. 633
reverse-vector->bitvector ................................ 378
reverse-vector->generator ................................ 437
reverse-vector->list .................................... 602
reverse-vector->accumulator ............................. 623
reverse-bitvector->vector/int ......................... 378
rfc822-atom .......................................... 919
rfc822-date->date ..................................... 919
rfc822-dot-atom ....................................... 919
rfc822-field->tokens .................................. 919
rfc822-header->1list .................................. 917
rfc822-header-ref ..................................... 917
rfc822-next-token ..................................... 918
rfc822-parse-date ..................................... 919
rfc822-quoted-string .................................. 919
rfc822-read-headers ................................... 917
rfc822-skip-cfws ...................................... 919
right .................................................. 766
right-section ......................................... 806
right? .................................................. 765
ring-buffer->flat-vector ................................ 851
ring-buffer:add-back! .................................. 851
ring-buffer:add-front! .................................. 851
ring-buffer-back ....................................... 851
ring-buffer-capacity .................................... 850
ring-buffer-empty? ..................................... 850
ring-buffer-front ...................................... 851
ring-buffer-full? ...................................... 850
ring-buffer-num-entries ................................. 850
ring-buffer-ref ........................................ 851
ring-buffer-remove-back! ............................... 851
ring-buffer-remove-front! ............................... 851
ring-buffer-set! ....................................... 851
rlet1 .................................................... 60
rope->string ........................................... 912
rope-finalize .......................................... 912
rotate! ................................................. 54
rotate-bit-field ........................................ 719
round ................................................... 131
round->exact .......................................... 131
round-quotient ........................................ 663
round-remainder ........................................ 663
round/ ................................................ 663
rtd-accessor .......................................... 507
rtd-all-field-names .................................... 506
rtd-constructor ........................................ 506
rtd-field-mutable? ..................................... 506
rtd-field-names ........................................ 506
rtd-mutator ............................................ 507
rtd-name ................................................ 506
rtd-parent ............................................. 506
rtd-predicate .......................................... 506
rtd? ..................................................... 506
run-cgi-script->header&body ........................... 1052
run-cgi-script->string ................................ 1053
run-cgi-script->string-list ............................. 1053
run-cgi-script->sxml .................................. 1053
run-hook ................................................ 448
run-pipeline .......................................... 494
run-process .......................................... 488
runtime ................................................ 787
rx ....................................................... 649
rxmatch ................................................. 187
rxmatch->string ........................................ 188
rxmatch->after ......................................... 188
rxmatch->before ........................................ 188
rxmatch->case .......................................... 192
rxmatch->cond .......................................... 192
rxmatch-end ............................................ 188
rxmatch-if ............................................. 191
rxmatch-let ............................................ 191
rxmatch->named-groups ................................. 189
rxmatch->num-matches ................................. 189
rxmatch->positions .................................... 188
rxmatch->start ......................................... 188
rxmatch->substring .................................... 188
rxmatch->substrings ................................... 188

S
s16? ...................................................... 557
s16array ............................................... 368
s16ite->u8vector ....................................... 568
s16ke->u8vector->generator ............................ 562
s16vector .............................................. 557
s16vector->1list ....................................... 563
s16vector->vector ...................................... 564
s16vector-add .......................................... 565
s16vector-add! .......................................... 565
s16vector-and .......................................... 566
s16vector-and! ......................................... 566
s16vector-any .......................................... 562
s16vector-append ...................................... 561
s16vector-append-subvectors ............................ 561
Appendix B: Function and Syntax Index

s16vector-clamp
s16vector-clamp!
s16vector-compare
s16vector-concatenate
s16vector-copy
s16vector-copy!
s16vector-count
s16vector-cumulate
s16vector-dot
s16vector-drop
s16vector-drop-right
s16vector-drop-while
s16vector-drop-while-right
s16vector-empty?
s16vector.every!

s16vector-fill!

s16vector-filter
s16vector-fold
s16vector-for-each
s16vector-index
s16vector-index-right
s16vector-ior
s16vector-ior!

s16vector-length
s16vector-map
s16vector-map!

s16vector-mul
s16vector-mul!

s16vector-multi-copy!

s16vector-partition
s16vector-range-check
s16vector-ref
s16vector-remove
s16vector-reverse!

s16vector-reverse-copy!

s16vector-set!

s16vector-skip!

s16vector-skip-right!

s16vector-sub!

s16vector-sub!

s16vector-swap!

s16vector-take
s16vector-take-right

s16vector-take-while

s16vector-take-while-right

s16vector-unfold!

s16vector-unfold-right!

s16vector-unfold-right!

s16vector-unfold-right!

s16vector-xor!

s16vector=!

s16vector=?

s32array
s32ite-u8vector
s32ite-u8vector-generator
s32vector
s32vector->list
s32vector->string

s32vector->vector

s32vector.add
s32vector.add!

s32vector.and

s32vector.append
s32vector.append-subvectors

s32vector.cumulate

s32vector.compare

s32vector.clamp

s32vector.copy

s32vector.copy!

s32vector.count

s32vector.cumulate

s32vector.clamp!

s32vector.drop

s32vector.drop-right

s32vector.drop-right!

s32vector.dot

s32vector.fold

s32vector.for-each

s32vector.index

s32vector.index-right

s32vector.ior

s32vector.ior!

s32vector.length

s32vector.map

s32vector.map!

s32vector.mul

s32vector.muli

s32vector.multi-copy!

s32vector.partition

s32vector.range-check

s32vector.ref

s32vector.remove

s32vector.reverse!

s32vector.reverse-copy!

s32vector.segment

s32vector.set!

s32vector.skip!

s32vector.skip-right!

s32vector.sub!

s32vector.sub!

s32vector.swap!

s32vector.take

s32vector.take-right!

s32vector.take-while!

s32vector.take-while-right!

s32vector.unfold!

s32vector.unfold-right!

s32vector.unfold-right!

s32vector.unfold-right!

s32vector.xor!

s32vector=x!

s32vector=?

s32vector?
stream-first......................... 1040  string->regexp.......................... 186
stream-fold........................... 1038  string->s32vector........................ 202
stream-for-each....................... 1038  string->s32vector!........................ 202
stream-format......................... 1036  string->s8vector........................ 201
stream-fourth......................... 1040  string->s8vector!........................ 202
stream-from......................... 1035  string->stream.......................... 1036
stream-grep........................... 1043  string->symbol.......................... 156
stream-index......................... 1043  string->text............................. 628
stream-intersperse.................... 1041  string->u32vector........................ 202
stream-iota........................... 1035  string->u32vector!........................ 202
stream-iterate......................... 1035  string->u8vector........................ 201
stream-lambda......................... 1033  string->u8vector!........................ 202
stream-last......................... 1041  string->uninterned-symbol.............. 156
stream-last-n........................ 1041  string->utf16........................... 552
stream-length......................... 1041  string->utf32........................... 552
stream-length=........................ 1041  string->utf8........................... 551
stream-length>=....................... 1041  string->vector.......................... 196
stream-let.......................... 1037  string->words........................... 553
stream-lines......................... 1039  string-accumulator...................... 633
stream-map........................... 1039  string-any.............................. 693
stream-match......................... 1037  string-append.......................... 178
stream-member......................... 1043  string-append!......................... 736, 764
stream-memq........................... 1043  string-append-ec....................... 712
stream-memv........................... 1043  string-append-linear!.................. 764
stream-ninth......................... 1040  string-append/shared................... 699
stream-null?......................... 787, 1033  string-break............................ 742
stream-of......................... 1037  string-build-index!..................... 176
stream-pair?......................... 1033  string-byte-ref........................ 177
stream-partition...................... 1039  string-byte-set!....................... 177
stream-prefix=....................... 1040  string-ci-hash.......................... 115, 621, 723
stream-range......................... 1035  string-ci<............................. 696
stream-ref......................... 1040  string-ci<=.............................. 696
stream-remove......................... 1039  string-ci=?............................ 178, 555
stream-reverse......................... 1041  string-ci<>............................ 696
stream-scan......................... 1039  string-ci?<............................. 178, 555
stream-second......................... 1040  string-ci=.............................. 696
stream-seventh......................... 1040  string-ci=?............................ 178, 555
stream-sixth......................... 1040  string-ci>.............................. 696
stream-span......................... 1041  string-ci>=.............................. 696
stream-split......................... 1041  string-ci>=?............................ 178, 555
stream-tabulate......................... 1035  string-ci?>............................ 178, 555
stream-take....................... 636, 1040  string-ci-hash!........................ 694
stream-take-safe...................... 1040  string-ci-hash!........................ 694
stream-take-while..................... 1042  string-compare.......................... 695
stream-tenth......................... 1040  string-compare-ci....................... 695
stream-third......................... 1040  string-concatenate..................... 698
stream-unfold......................... 1034  string-concatenate-reverse............ 699
stream-unfoldn......................... 1034  string-concatenate-reverse/shared..... 699
stream-unfolds......................... 1035  string-concatenate/shared.............. 699
stream-xcons......................... 1036  string-contains........................ 698, 740
stream-zip......................... 1039  string-contains-ci....................... 698
stream?.......................... 1033  string-contains-right.................. 740, 741
string.......................... 172  string-copy............................... 179
string->bitvector..................... 204  string-copy!.............................. 694
string->bytevector.................... 763  string-copy-immutable................... 179
string->char-set...................... 614  string-count....................... 697
string->char-set!..................... 614  string-cursor->index.................... 176
string->date......................... 706  string-cursor->index!................... 176
string->file......................... 890  string-cursor-back...................... 175
string->gap-buffer.................... 998  string-cursor-diff....................... 176
string->generator..................... 437  string-cursor-end....................... 175
string->grapheme-clusters........... 553  string-cursor-forward.................. 175
string->list......................... 178  string-cursor-next.................... 175
string->number......................... 136  string-cursor-prev..................... 175
string->number!........................ 136  string-cursor-start................... 175
string->number?..................... 176  string-cursor=?......................... 176
string->number?........................ 176
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys-get-conorder-cp</td>
<td>899</td>
</tr>
<tr>
<td>sys-get-conorder-cursor-info</td>
<td>899</td>
</tr>
<tr>
<td>sys-get-conorder-mode</td>
<td>899</td>
</tr>
<tr>
<td>sys-get-conorder-output-cp</td>
<td>899</td>
</tr>
<tr>
<td>sys-get-conorder-screen-buffer-info</td>
<td>900</td>
</tr>
<tr>
<td>sys-get-conorder-title</td>
<td>901</td>
</tr>
<tr>
<td>sys-get-largest-conorder-window-size</td>
<td>900</td>
</tr>
<tr>
<td>sys-get-number-of-conorder-input-events</td>
<td>901</td>
</tr>
<tr>
<td>sys-get-number-of-conorder-mouse-buttons</td>
<td>901</td>
</tr>
<tr>
<td>sys-get-oshandle</td>
<td>326</td>
</tr>
<tr>
<td>sys-get-std-handle</td>
<td>902</td>
</tr>
<tr>
<td>sys-getaddrinfo</td>
<td>477</td>
</tr>
<tr>
<td>sys-getcwd</td>
<td>313</td>
</tr>
<tr>
<td>sys-getdomainname</td>
<td>313</td>
</tr>
<tr>
<td>sys-getegid</td>
<td>313</td>
</tr>
<tr>
<td>sys-getenv</td>
<td>295</td>
</tr>
<tr>
<td>sys-geteuid</td>
<td>313</td>
</tr>
<tr>
<td>sys-getgid</td>
<td>313</td>
</tr>
<tr>
<td>sys-getggrid</td>
<td>305</td>
</tr>
<tr>
<td>sys-getgrman</td>
<td>305</td>
</tr>
<tr>
<td>sys-getgroups</td>
<td>314</td>
</tr>
<tr>
<td>sys-gethostbyaddr</td>
<td>475</td>
</tr>
<tr>
<td>sys-gethostbyname</td>
<td>313</td>
</tr>
<tr>
<td>sys-gethostname</td>
<td>313</td>
</tr>
<tr>
<td>sys-getlogin</td>
<td>314</td>
</tr>
<tr>
<td>sys-getpgid</td>
<td>314</td>
</tr>
<tr>
<td>sys-getppid</td>
<td>314</td>
</tr>
<tr>
<td>sys-getppid</td>
<td>314</td>
</tr>
<tr>
<td>sys-getprotobyname</td>
<td>476</td>
</tr>
<tr>
<td>sys-getprobobynumber</td>
<td>477</td>
</tr>
<tr>
<td>sys-getpnum</td>
<td>305</td>
</tr>
<tr>
<td>sys-getpuid</td>
<td>305</td>
</tr>
<tr>
<td>sys-getrlimit</td>
<td>315</td>
</tr>
<tr>
<td>sys-getservbyname</td>
<td>476</td>
</tr>
<tr>
<td>sys-getsevbyport</td>
<td>476</td>
</tr>
<tr>
<td>sys-gettimeofday</td>
<td>316</td>
</tr>
<tr>
<td>sys-getuid</td>
<td>313</td>
</tr>
<tr>
<td>sys-gid-&gt;group-name</td>
<td>297</td>
</tr>
<tr>
<td>sys-gmtime</td>
<td>317</td>
</tr>
<tr>
<td>sys-group-name-&gt;gid</td>
<td>305</td>
</tr>
<tr>
<td>sys-htol</td>
<td>477</td>
</tr>
<tr>
<td>sys-htons</td>
<td>477</td>
</tr>
<tr>
<td>sys-icatty</td>
<td>304</td>
</tr>
<tr>
<td>sys-kill</td>
<td>308</td>
</tr>
<tr>
<td>sys-link</td>
<td>299</td>
</tr>
<tr>
<td>sys/localeconv</td>
<td>306</td>
</tr>
<tr>
<td>sys-localtime</td>
<td>317</td>
</tr>
<tr>
<td>sys-logmask</td>
<td>518</td>
</tr>
<tr>
<td>sys-lstat</td>
<td>302</td>
</tr>
<tr>
<td>sys-messagebox</td>
<td>898</td>
</tr>
<tr>
<td>sys-mkdir</td>
<td>300</td>
</tr>
<tr>
<td>sys-mktemp</td>
<td>299</td>
</tr>
<tr>
<td>sys-mkfifo</td>
<td>304</td>
</tr>
<tr>
<td>sys-mktemp</td>
<td>299</td>
</tr>
<tr>
<td>sys-mtime</td>
<td>317</td>
</tr>
<tr>
<td>sys-map</td>
<td>323</td>
</tr>
<tr>
<td>sys-nanosleep</td>
<td>325</td>
</tr>
<tr>
<td>sys-nice</td>
<td>315</td>
</tr>
<tr>
<td>sys-normalize-pathname</td>
<td>300</td>
</tr>
<tr>
<td>sys-ntho1</td>
<td>477</td>
</tr>
<tr>
<td>sys-ntho2</td>
<td>477</td>
</tr>
<tr>
<td>sys-open</td>
<td>344</td>
</tr>
<tr>
<td>sys-openlog</td>
<td>518</td>
</tr>
</tbody>
</table>
sys-statvfs ........................................ 435
sys-strerror ......................................... 315
sys-strftime ......................................... 317
sys-strsignal .......................................... 308
sys-symbol->errno .................................... 316
sys-symlink ........................................... 299
sys-syslog ............................................ 518
sys-system ............................................. 318
sys-tcdrain ........................................... 519
sys-tcflow ............................................ 520
sys-tcflush ........................................... 519
sys-tcgetattr .......................................... 519
sys-tcgetpgrp ........................................ 520
sys-tcsendbreak ...................................... 519
sys-tcssetattr ........................................ 519
sys-tcssetpgrp ........................................ 520
sys-time ................................................ 316
sys-times .............................................. 314
sys-tm->alist ........................................... 317
sys-tmpdir ............................................ 301
sys-tmpnam ............................................ 299
sys-truncate ........................................... 304
sys-tyname ............................................ 304
sys-uid->user-name ..................................... 305
sys-umask ............................................... 300
sys-uname .............................................. 313
sys-unlink ............................................. 299
sys-unsetenv .......................................... 296
sys-user-name->uid ..................................... 303
sys-utime .............................................. 303
sys-wait ................................................ 320
sys-wait-exit-status .................................. 321
sys-wait-exited? ....................................... 321
sys-wait-signaled? ..................................... 321
sys-wait-stopped? ...................................... 321
sys-wait-stopsig ....................................... 321
sys-wait-term.sig ...................................... 321
sys-waitpid ............................................ 320
sys-win-process-pid ................................... 321
sys-win-process? ....................................... 321
sys-write-console ...................................... 901
sys-write-console-output-character ............... 901
terminate-all! .......................................... 827
terminated-thread-exception? ......................... 545
test ....................................................... 524
test* ..................................................... 524
test*/diff ............................................... 528
test-and-set! .......................................... 787
test-check .............................................. 525
test-check-diff ......................................... 528
test-end ................................................. 522
test-error ............................................... 526
test-include-r7 ......................................... 527
test-log .................................................. 522
test-module .............................................. 529
test-none-of ............................................ 525
test-one-of ............................................. 525
test-record-file ......................................... 522
test-report-failure ..................................... 525
test-report-failure-diff ............................... 528
test-script ............................................. 529
test-secion .............................................. 522
test-start ............................................... 522
test-summary-check ..................................... 523
test-truthy .............................................. 526
text ....................................................... 627
text-length ............................................. 627
text-ref .................................................. 627
text-tabulate ............................................ 627
text-unfold ............................................. 627
text-unfold-right ....................................... 627
text? ....................................................... 627
textdomain .............................................. 1000
textual->list ............................................ 628
textual->string ......................................... 628
textual->text ............................................ 628
textual->utf16 .......................................... 628
textual->utf16be ......................................... 628
textual->utf16le ......................................... 628
textual->utf8 ........................................... 628
textual->vector ......................................... 627
textual-any ............................................. 627
textual-append ......................................... 630
textual-ci=? ............................................ 630
textual-ci? ............................................. 630
textual-ci= ............................................. 630
textual-ci>= ............................................ 630
textual-concatenate ..................................... 630
textual-concatenate-reverse ................................ 630
textual-contains ....................................... 630
textual-contains-right .................................. 630
textual-copy ............................................. 630
textual-count ............................................ 631
textual-downcase ....................................... 630
textual-drop ............................................. 629
textual-drop-right ...................................... 629
textual-every ........................................... 627
textual-filter ........................................... 631
textual-fold ............................................. 631
textual-fold-right ...................................... 631
textual-foldcase ........................................ 630
textual-for-each ......................................... 631
textual-for-each-index ................................... 631
textual-index ............................................ 630
textual-index-right ..................................... 630
textual-join ............................................ 631
Appendix B: Function and Syntax Index 1113

<table>
<thead>
<tr>
<th>Textual Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>textual-length</td>
<td>628</td>
</tr>
<tr>
<td>textual-map</td>
<td>631</td>
</tr>
<tr>
<td>textual-map-index</td>
<td>631</td>
</tr>
<tr>
<td>textual-null?</td>
<td>627</td>
</tr>
<tr>
<td>textual-pad</td>
<td>629</td>
</tr>
<tr>
<td>textual-pad-right</td>
<td>629</td>
</tr>
<tr>
<td>textual-port?</td>
<td>587</td>
</tr>
<tr>
<td>textual-prefix-length</td>
<td>630</td>
</tr>
<tr>
<td>textual-prefix?</td>
<td>630</td>
</tr>
<tr>
<td>textual-ref</td>
<td>628</td>
</tr>
<tr>
<td>textual-remove</td>
<td>631</td>
</tr>
<tr>
<td>textual-replace</td>
<td>629</td>
</tr>
<tr>
<td>textual-replicate</td>
<td>611</td>
</tr>
<tr>
<td>textual-skip</td>
<td>630</td>
</tr>
<tr>
<td>textual-skip-right</td>
<td>630</td>
</tr>
<tr>
<td>textual-split</td>
<td>631</td>
</tr>
<tr>
<td>textual-suffix-length</td>
<td>630</td>
</tr>
<tr>
<td>textual-suffix?</td>
<td>630</td>
</tr>
<tr>
<td>textual-take</td>
<td>629</td>
</tr>
<tr>
<td>textual-take-right</td>
<td>629</td>
</tr>
<tr>
<td>textual-titlecase</td>
<td>630</td>
</tr>
<tr>
<td>textual-trim</td>
<td>629</td>
</tr>
<tr>
<td>textual-trim-both</td>
<td>629</td>
</tr>
<tr>
<td>textual-trim-right</td>
<td>629</td>
</tr>
<tr>
<td>textual-upcase</td>
<td>630</td>
</tr>
<tr>
<td>textual=?</td>
<td>629</td>
</tr>
<tr>
<td>textual&lt;?</td>
<td>629</td>
</tr>
<tr>
<td>textual?&gt;</td>
<td>629</td>
</tr>
<tr>
<td>textual&gt;=?</td>
<td>629</td>
</tr>
<tr>
<td>textual?</td>
<td>627</td>
</tr>
<tr>
<td>third</td>
<td>594</td>
</tr>
<tr>
<td>thread-abandoned-mutex-condition?</td>
<td>545</td>
</tr>
<tr>
<td>thread-already-terminated-condition?</td>
<td>545</td>
</tr>
<tr>
<td>thread-condition?</td>
<td>545</td>
</tr>
<tr>
<td>thread-cont!</td>
<td>534</td>
</tr>
<tr>
<td>thread-join!</td>
<td>535</td>
</tr>
<tr>
<td>thread-local?</td>
<td>536</td>
</tr>
<tr>
<td>thread-name</td>
<td>536</td>
</tr>
<tr>
<td>thread-pool-results</td>
<td>826</td>
</tr>
<tr>
<td>thread-pool-shut-down?</td>
<td>826</td>
</tr>
<tr>
<td>thread-sleep!</td>
<td>534</td>
</tr>
<tr>
<td>thread-specific</td>
<td>533</td>
</tr>
<tr>
<td>thread-specific-set!</td>
<td>533</td>
</tr>
<tr>
<td>thread-start!</td>
<td>534</td>
</tr>
<tr>
<td>thread-state</td>
<td>533</td>
</tr>
<tr>
<td>thread-stop!</td>
<td>534</td>
</tr>
<tr>
<td>thread-terminate!</td>
<td>535</td>
</tr>
<tr>
<td>thread-timeout-condition?</td>
<td>545</td>
</tr>
<tr>
<td>thread-try-start!</td>
<td>534</td>
</tr>
<tr>
<td>thread-yield!</td>
<td>534</td>
</tr>
<tr>
<td>thread?</td>
<td>533</td>
</tr>
<tr>
<td>time</td>
<td>546</td>
</tr>
<tr>
<td>time-&gt;seconds</td>
<td>318</td>
</tr>
<tr>
<td>time-counter-reset!</td>
<td>549</td>
</tr>
<tr>
<td>time-counter-start!</td>
<td>548</td>
</tr>
<tr>
<td>time-counter-stop!</td>
<td>548</td>
</tr>
<tr>
<td>time-counter-value</td>
<td>549</td>
</tr>
<tr>
<td>time-difference</td>
<td>703</td>
</tr>
<tr>
<td>time-difference!</td>
<td>703</td>
</tr>
<tr>
<td>time-monotonic-&gt;date</td>
<td>704</td>
</tr>
<tr>
<td>time-monotonic-&gt;time-utc</td>
<td>704</td>
</tr>
<tr>
<td>time-monotonic-&gt;time-utc!</td>
<td>704</td>
</tr>
<tr>
<td>time-nanosecond</td>
<td>702</td>
</tr>
<tr>
<td>time-resolution</td>
<td>702</td>
</tr>
<tr>
<td>time-result+</td>
<td>547</td>
</tr>
<tr>
<td>time-result-</td>
<td>547</td>
</tr>
<tr>
<td>time-result-count</td>
<td>547</td>
</tr>
<tr>
<td>time-result-real</td>
<td>547</td>
</tr>
<tr>
<td>time-result-sys</td>
<td>547</td>
</tr>
<tr>
<td>time-result-user</td>
<td>547</td>
</tr>
<tr>
<td>time-result?</td>
<td>547</td>
</tr>
<tr>
<td>time-second</td>
<td>702</td>
</tr>
<tr>
<td>time-tai-&gt;date</td>
<td>705</td>
</tr>
<tr>
<td>time-tai-&gt;julian-day</td>
<td>705</td>
</tr>
<tr>
<td>time-tai-&gt;modified-julian-day</td>
<td>705</td>
</tr>
<tr>
<td>time-tai-&gt;time-monotonic</td>
<td>705</td>
</tr>
<tr>
<td>time-tai-&gt;time-monotonic!</td>
<td>705</td>
</tr>
<tr>
<td>time-tai-&gt;time-utc</td>
<td>705</td>
</tr>
<tr>
<td>time-tai-&gt;time-utc!</td>
<td>705</td>
</tr>
<tr>
<td>time-these</td>
<td>547</td>
</tr>
<tr>
<td>time-these/report</td>
<td>547</td>
</tr>
<tr>
<td>time-this</td>
<td>546</td>
</tr>
<tr>
<td>time-type</td>
<td>702</td>
</tr>
<tr>
<td>time-utc-&gt;date</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;julian-day</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;modified-julian-day</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;time-monotonic</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;time-monotonic!</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;time-tai</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;time-tai!</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;time-utc</td>
<td>705</td>
</tr>
<tr>
<td>time-utc-&gt;time-utc!</td>
<td>705</td>
</tr>
<tr>
<td>timer?</td>
<td>703</td>
</tr>
<tr>
<td>time&gt;=?</td>
<td>703</td>
</tr>
<tr>
<td>time&gt;?</td>
<td>703</td>
</tr>
<tr>
<td>time=</td>
<td>703</td>
</tr>
<tr>
<td>time-?</td>
<td>703</td>
</tr>
<tr>
<td>time?</td>
<td>318</td>
</tr>
<tr>
<td>timer-cancel!</td>
<td>737</td>
</tr>
<tr>
<td>timer-delta?</td>
<td>738</td>
</tr>
<tr>
<td>timer-reschedule!</td>
<td>739</td>
</tr>
<tr>
<td>timer-schedule!</td>
<td>737</td>
</tr>
<tr>
<td>timer-task-exists?</td>
<td>738</td>
</tr>
<tr>
<td>timer-task-remove!</td>
<td>737</td>
</tr>
<tr>
<td>timer?</td>
<td>737</td>
</tr>
<tr>
<td>timespec</td>
<td>752</td>
</tr>
<tr>
<td>timespec-&gt;inexact</td>
<td>752</td>
</tr>
<tr>
<td>timespec-hash</td>
<td>753</td>
</tr>
<tr>
<td>timespec-nanosecods</td>
<td>752</td>
</tr>
<tr>
<td>timespec-seconds</td>
<td>752</td>
</tr>
<tr>
<td>timespec?</td>
<td>752</td>
</tr>
<tr>
<td>timespec?</td>
<td>752</td>
</tr>
<tr>
<td>tlref</td>
<td>536</td>
</tr>
<tr>
<td>tls-accept</td>
<td>948</td>
</tr>
<tr>
<td>tls-ca-bundle-path</td>
<td>947</td>
</tr>
<tr>
<td>tls-close</td>
<td>949</td>
</tr>
<tr>
<td>tls-connect</td>
<td>948</td>
</tr>
<tr>
<td>tls-destroy</td>
<td>949</td>
</tr>
<tr>
<td>tls-input-port</td>
<td>949</td>
</tr>
<tr>
<td>tls-load-object</td>
<td>949</td>
</tr>
<tr>
<td>tls-output-port</td>
<td>949</td>
</tr>
<tr>
<td>tlet!</td>
<td>536</td>
</tr>
<tr>
<td>topological-sort</td>
<td>1046</td>
</tr>
<tr>
<td>totient</td>
<td>897</td>
</tr>
<tr>
<td>touch-file</td>
<td>888</td>
</tr>
<tr>
<td>touch-files</td>
<td>888</td>
</tr>
</tbody>
</table>
### Appendix B: Function and Syntax Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>trie-values</td>
<td>862</td>
</tr>
<tr>
<td>trie-with-keys</td>
<td>861</td>
</tr>
<tr>
<td>trie?</td>
<td>861</td>
</tr>
<tr>
<td>trimmed</td>
<td>685</td>
</tr>
<tr>
<td>trimmed/both</td>
<td>685</td>
</tr>
<tr>
<td>trimmed/lazy</td>
<td>685</td>
</tr>
<tr>
<td>trimmed/right</td>
<td>685</td>
</tr>
<tr>
<td>truncate</td>
<td>131</td>
</tr>
<tr>
<td>truncate-&gt;exact</td>
<td>131</td>
</tr>
<tr>
<td>truncate-file</td>
<td>746</td>
</tr>
<tr>
<td>truncate-quotient</td>
<td>129</td>
</tr>
<tr>
<td>truncate-remainder</td>
<td>129</td>
</tr>
<tr>
<td>truth-&gt;either</td>
<td>769</td>
</tr>
<tr>
<td>truth-&gt;maybe</td>
<td>769</td>
</tr>
<tr>
<td>trie-and</td>
<td>772</td>
</tr>
<tr>
<td>try-merge</td>
<td>773</td>
</tr>
<tr>
<td>try-or</td>
<td>772</td>
</tr>
<tr>
<td>try=?</td>
<td>772</td>
</tr>
<tr>
<td>tuples-of</td>
<td>844</td>
</tr>
<tr>
<td>two-values-&gt;maybe</td>
<td>770</td>
</tr>
<tr>
<td>two-exponent</td>
<td>138</td>
</tr>
<tr>
<td>two-exponent-factor</td>
<td>138</td>
</tr>
<tr>
<td>typecase</td>
<td>56</td>
</tr>
<tr>
<td>u16?</td>
<td>557</td>
</tr>
<tr>
<td>u8vector</td>
<td>568</td>
</tr>
<tr>
<td>u8vector-&gt;generator</td>
<td>562</td>
</tr>
<tr>
<td>u8vector</td>
<td>557</td>
</tr>
<tr>
<td>u8vector-&gt;list</td>
<td>563</td>
</tr>
<tr>
<td>u8vector-&gt;vector</td>
<td>564</td>
</tr>
<tr>
<td>u8vector-&gt;add!</td>
<td>565</td>
</tr>
<tr>
<td>u8vector-&gt;add</td>
<td>565</td>
</tr>
<tr>
<td>u8vector-&gt;and</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-&gt;and!</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-&gt;any</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-append</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-append-subvectors</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-clamp</td>
<td>567</td>
</tr>
<tr>
<td>u8vector-clamp!</td>
<td>567</td>
</tr>
<tr>
<td>u8vector-compare</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-concatenate</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-copy</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-count</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-cumulate</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-dot</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-drop</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-drop-right</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-drop-while</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-drop-while-right</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-empty?</td>
<td>557</td>
</tr>
<tr>
<td>u8vector-every</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-fill</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-filter</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-fold</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-for-each</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-index</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-ior</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-ior!</td>
<td>566</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>u16vector-length</td>
<td>558</td>
</tr>
<tr>
<td>u16vector-map</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-map!</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-mul</td>
<td>565</td>
</tr>
<tr>
<td>u16vector-mul!</td>
<td>565</td>
</tr>
<tr>
<td>u16vector-multi-copy</td>
<td>559</td>
</tr>
<tr>
<td>u16vector-partition</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-range-check</td>
<td>566</td>
</tr>
<tr>
<td>u16vector-ref</td>
<td>201</td>
</tr>
<tr>
<td>u16vector-remove</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-reverse!</td>
<td>559</td>
</tr>
<tr>
<td>u16vector-reverse-copy</td>
<td>559</td>
</tr>
<tr>
<td>u16vector-reverse-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u16vector-segment</td>
<td>561</td>
</tr>
<tr>
<td>u16vector-set!</td>
<td>201</td>
</tr>
<tr>
<td>u16vector-skip</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-skip-right</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-sub</td>
<td>565</td>
</tr>
<tr>
<td>u16vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>u16vector-swap!</td>
<td>558</td>
</tr>
<tr>
<td>u16vector-take</td>
<td>561</td>
</tr>
<tr>
<td>u16vector-take-right</td>
<td>561</td>
</tr>
<tr>
<td>u16vector-take-while</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-take-while-right</td>
<td>562</td>
</tr>
<tr>
<td>u16vector-unfold</td>
<td>557, 561</td>
</tr>
<tr>
<td>u16vector-unfold!</td>
<td>557, 561</td>
</tr>
<tr>
<td>u16vector-unfold-right</td>
<td>557</td>
</tr>
<tr>
<td>u16vector-unfold-right!</td>
<td>557</td>
</tr>
<tr>
<td>u16vector-xor</td>
<td>566</td>
</tr>
<tr>
<td>u16vector-xor!</td>
<td>566</td>
</tr>
<tr>
<td>u16vector=</td>
<td>558</td>
</tr>
<tr>
<td>u16vector=!?</td>
<td>558</td>
</tr>
<tr>
<td>u16vector?</td>
<td>200</td>
</tr>
<tr>
<td>u32?</td>
<td>557</td>
</tr>
<tr>
<td>u32array</td>
<td>368</td>
</tr>
<tr>
<td>u32ite-u8vector</td>
<td>568</td>
</tr>
<tr>
<td>u32ke-u8vector-generator</td>
<td>562</td>
</tr>
<tr>
<td>u32vector</td>
<td>561</td>
</tr>
<tr>
<td>u32vector&gt;!</td>
<td>563</td>
</tr>
<tr>
<td>u32vector-&gt;string</td>
<td>203</td>
</tr>
<tr>
<td>u32vector-&gt;vector</td>
<td>564</td>
</tr>
<tr>
<td>u32vector-add</td>
<td>565</td>
</tr>
<tr>
<td>u32vector-add!</td>
<td>565</td>
</tr>
<tr>
<td>u32vector-add!</td>
<td>565</td>
</tr>
<tr>
<td>u32vector-and</td>
<td>566</td>
</tr>
<tr>
<td>u32vector-and!</td>
<td>566</td>
</tr>
<tr>
<td>u32vector-any</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-append</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-append-subvectors</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-assign</td>
<td>567</td>
</tr>
<tr>
<td>u32vector-assign!</td>
<td>567</td>
</tr>
<tr>
<td>u32vector-compare</td>
<td>558</td>
</tr>
<tr>
<td>u32vector-concatenate</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-copy</td>
<td>559</td>
</tr>
<tr>
<td>u32vector-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u32vector-count</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-cumulate</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-dot</td>
<td>566</td>
</tr>
<tr>
<td>u32vector-drop</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-drop-right</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-drop-while</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-drop-while-right</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-empty!</td>
<td>557</td>
</tr>
<tr>
<td>u32vector-every!</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-fill!</td>
<td>558</td>
</tr>
<tr>
<td>u32vector-filter</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-fold</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-for-each</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-index</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-ior</td>
<td>566</td>
</tr>
<tr>
<td>u32vector-index</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-length</td>
<td>558</td>
</tr>
<tr>
<td>u32vector-map</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-map!</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-mul</td>
<td>565</td>
</tr>
<tr>
<td>u32vector-mul!</td>
<td>565</td>
</tr>
<tr>
<td>u32vector-multi-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u32vector-partition</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-set</td>
<td>201</td>
</tr>
<tr>
<td>u32vector-remove</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-reverse</td>
<td>559</td>
</tr>
<tr>
<td>u32vector-reverse-copy</td>
<td>559</td>
</tr>
<tr>
<td>u32vector-reverse-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u32vector-segment</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-set!</td>
<td>201</td>
</tr>
<tr>
<td>u32vector-skip</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-skip-right</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-take</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-take-right</td>
<td>561</td>
</tr>
<tr>
<td>u32vector-take-while</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-take-while-right</td>
<td>562</td>
</tr>
<tr>
<td>u32vector-unfold</td>
<td>557, 561</td>
</tr>
<tr>
<td>u32vector-unfold!</td>
<td>557, 561</td>
</tr>
<tr>
<td>u32vector-unfold-right</td>
<td>557</td>
</tr>
<tr>
<td>u32vector-unfold-right!</td>
<td>557</td>
</tr>
<tr>
<td>u32vector-xor</td>
<td>566</td>
</tr>
<tr>
<td>u32vector-xor!</td>
<td>566</td>
</tr>
<tr>
<td>u32vector-?</td>
<td>558</td>
</tr>
<tr>
<td>u32vector?</td>
<td>200</td>
</tr>
<tr>
<td>u64?</td>
<td>557</td>
</tr>
<tr>
<td>u64vector</td>
<td>564</td>
</tr>
<tr>
<td>u64vector-&gt;list</td>
<td>563</td>
</tr>
<tr>
<td>u64vector-&gt;vector</td>
<td>564</td>
</tr>
<tr>
<td>u64vector-append</td>
<td>561</td>
</tr>
<tr>
<td>u64vector-append-subvectors</td>
<td>561</td>
</tr>
<tr>
<td>u64vector-assign</td>
<td>567</td>
</tr>
<tr>
<td>u64vector-any</td>
<td>562</td>
</tr>
<tr>
<td>u64vector-append</td>
<td>561</td>
</tr>
<tr>
<td>u64vector-append-subvectors</td>
<td>561</td>
</tr>
<tr>
<td>u64vector-assigned</td>
<td>566</td>
</tr>
<tr>
<td>u64vector-clamp</td>
<td>567</td>
</tr>
<tr>
<td>u64vector-clamp!</td>
<td>567</td>
</tr>
<tr>
<td>u64vector-compare</td>
<td>558</td>
</tr>
<tr>
<td>u64vector-copy</td>
<td>559</td>
</tr>
<tr>
<td>u64vector-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u64vector-count</td>
<td>562</td>
</tr>
<tr>
<td>u64vector-cumulate</td>
<td>562</td>
</tr>
<tr>
<td>u64vector-dot</td>
<td>566</td>
</tr>
<tr>
<td>u64vector-drop</td>
<td>561</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>u8vector-apply</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-any</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-and</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-&gt;string</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-&gt;array</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-&gt;list</td>
<td>563</td>
</tr>
<tr>
<td>u8vector-&gt;string</td>
<td>202</td>
</tr>
<tr>
<td>u8vector-&gt;vector</td>
<td>564</td>
</tr>
<tr>
<td>u8vector-add</td>
<td>565</td>
</tr>
<tr>
<td>u8vector-add!</td>
<td>565</td>
</tr>
<tr>
<td>u8vector-and</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-and!</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-any</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-append</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-append-subvectors</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-clamp</td>
<td>567</td>
</tr>
<tr>
<td>u8vector-clamp!</td>
<td>567</td>
</tr>
<tr>
<td>u8vector-compare</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-concatenate</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-copy</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-copy!</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-count</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-cumulate</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-dot</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-drop</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-drop-right</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-drop-right!</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-drop-while</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-drop-while-right</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-empty</td>
<td>557</td>
</tr>
<tr>
<td>u8vector-every</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-empty!</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-index</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-index-right</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-ior</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-ior!</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-length</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-length!</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-remove</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-remove!</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-reverse</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-reverse-copy</td>
<td>559</td>
</tr>
<tr>
<td>u8vector-segment</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-set!</td>
<td>201</td>
</tr>
<tr>
<td>u8vector-set-right</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-sub</td>
<td>565</td>
</tr>
<tr>
<td>u8vector-sub!</td>
<td>565</td>
</tr>
<tr>
<td>u8vector-swap!</td>
<td>558</td>
</tr>
<tr>
<td>u8vector-take</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-take-right</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-take-right!</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-take-while</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-take-while-right</td>
<td>562</td>
</tr>
<tr>
<td>u8vector-unfold-right</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-unfold-right!</td>
<td>561</td>
</tr>
<tr>
<td>u8vector-unfold</td>
<td>557, 561</td>
</tr>
<tr>
<td>u8vector-unfold-right</td>
<td>557</td>
</tr>
<tr>
<td>u8vector-unfold-right!</td>
<td>557</td>
</tr>
<tr>
<td>u8vector-xor</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-xor!</td>
<td>566</td>
</tr>
<tr>
<td>u8vector-?</td>
<td>558</td>
</tr>
<tr>
<td>u8vector?</td>
<td>558</td>
</tr>
<tr>
<td>u8list-&gt;blob</td>
<td>725</td>
</tr>
<tr>
<td>u8list-&gt;bytevector</td>
<td>570</td>
</tr>
<tr>
<td>u8-ready?</td>
<td>272</td>
</tr>
<tr>
<td>u8?</td>
<td>557</td>
</tr>
<tr>
<td>u8array-&gt;unfold</td>
<td>557, 561</td>
</tr>
<tr>
<td>u8vector=</td>
<td>558</td>
</tr>
<tr>
<td>u8vector=?</td>
<td>558</td>
</tr>
<tr>
<td>u8vector?</td>
<td>200</td>
</tr>
<tr>
<td>u8vector?</td>
<td>163</td>
</tr>
<tr>
<td>ucs-&gt;char</td>
<td>163</td>
</tr>
<tr>
<td>ucs-&gt;char-set</td>
<td>614</td>
</tr>
<tr>
<td>ucs-range-&gt;char-set</td>
<td>614</td>
</tr>
<tr>
<td>ucs-range-&gt;char-set!</td>
<td>614</td>
</tr>
<tr>
<td>ucs4-&gt;utf16</td>
<td>551</td>
</tr>
<tr>
<td>ucs4-&gt;utf8</td>
<td>550</td>
</tr>
<tr>
<td>uint-list-&gt;blob</td>
<td>725</td>
</tr>
<tr>
<td>uint-list-&gt;bytevector</td>
<td>680</td>
</tr>
<tr>
<td>uint16s</td>
<td>842</td>
</tr>
<tr>
<td>uint32s</td>
<td>842</td>
</tr>
<tr>
<td>uint64s</td>
<td>842</td>
</tr>
</tbody>
</table>
Appendix B: Function and Syntax Index

V

valid-sre? ........................................ 649
valid-version-spec? .............................. 572
value ................................................. 785
value/procedure .................................. 806
value/mv ........................................... 785
values ............................................... 230
values->either ..................................... 770
values->list ....................................... 231
values->maybe .................................... 770
values-ref ........................................... 231
vector .................................................. 195
vector->@vector .................................... 564
vector->bits ........................................... 667
vector->bitvector .................................. 378
vector->c128vector .................................. 564
vector->c32vector .................................. 564
vector->c64vector .................................. 564
vector->f16vector .................................. 564
vector->f32vector .................................. 564
vector->f64vector .................................. 564
vector->generator ................................... 437
vector->list ......................................... 196
vector->range ......................................... 849

vi

V
<table>
<thead>
<tr>
<th>Function/Element</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrapped/char</td>
<td>686</td>
</tr>
<tr>
<td>wrapped/list</td>
<td>685</td>
</tr>
<tr>
<td>write</td>
<td>277</td>
</tr>
<tr>
<td>write*</td>
<td>278</td>
</tr>
<tr>
<td>write-bvector</td>
<td>568</td>
</tr>
<tr>
<td>write-ber-integer</td>
<td>809</td>
</tr>
<tr>
<td>write-binary-bitestring</td>
<td>779</td>
</tr>
<tr>
<td>write-block</td>
<td>568</td>
</tr>
<tr>
<td>write-byte</td>
<td>285</td>
</tr>
<tr>
<td>write-bytevector</td>
<td>568</td>
</tr>
<tr>
<td>write-char</td>
<td>285</td>
</tr>
<tr>
<td>write-controls-copy</td>
<td>277</td>
</tr>
<tr>
<td>write-f16</td>
<td>809</td>
</tr>
<tr>
<td>write-f32</td>
<td>809</td>
</tr>
<tr>
<td>write-f64</td>
<td>809</td>
</tr>
<tr>
<td>write-gauche-package-description</td>
<td>479</td>
</tr>
<tr>
<td>write-object</td>
<td>279</td>
</tr>
<tr>
<td>write-s16</td>
<td>809</td>
</tr>
<tr>
<td>write-s32</td>
<td>809</td>
</tr>
<tr>
<td>write-s64</td>
<td>809</td>
</tr>
<tr>
<td>write-s8</td>
<td>809</td>
</tr>
<tr>
<td>write-shared</td>
<td>277</td>
</tr>
<tr>
<td>write-simple</td>
<td>277</td>
</tr>
<tr>
<td>write-sint</td>
<td>809</td>
</tr>
<tr>
<td>write-stream</td>
<td>1043</td>
</tr>
<tr>
<td>write-string</td>
<td>285</td>
</tr>
<tr>
<td>write-textual-bytestring</td>
<td>778</td>
</tr>
<tr>
<td>write-to-string</td>
<td>270</td>
</tr>
<tr>
<td>write-tree</td>
<td>1015</td>
</tr>
<tr>
<td>write-u16</td>
<td>808</td>
</tr>
<tr>
<td>write-u32</td>
<td>808</td>
</tr>
<tr>
<td>write-u64</td>
<td>808</td>
</tr>
<tr>
<td>write-u8</td>
<td>285, 808</td>
</tr>
<tr>
<td>write-u8vector</td>
<td>568</td>
</tr>
<tr>
<td>write-uint</td>
<td>809</td>
</tr>
<tr>
<td>write-uuid</td>
<td>955</td>
</tr>
<tr>
<td>write-uvector</td>
<td>568</td>
</tr>
<tr>
<td>write-with-shared-structure</td>
<td>278</td>
</tr>
<tr>
<td>write/ss</td>
<td>278</td>
</tr>
<tr>
<td>written</td>
<td>682</td>
</tr>
<tr>
<td>written-simply</td>
<td>682</td>
</tr>
<tr>
<td>x-&gt;generator</td>
<td>439</td>
</tr>
<tr>
<td>x-&gt;integer</td>
<td>136</td>
</tr>
<tr>
<td>x-&gt;lseq</td>
<td>451</td>
</tr>
<tr>
<td>x-&gt;number</td>
<td>136</td>
</tr>
<tr>
<td>x-&gt;string</td>
<td>173</td>
</tr>
<tr>
<td>xcons</td>
<td>593</td>
</tr>
<tr>
<td>xml-token-head</td>
<td>962</td>
</tr>
<tr>
<td>xml-token-kind</td>
<td>962</td>
</tr>
<tr>
<td>xml-token?</td>
<td>962</td>
</tr>
<tr>
<td>xsubstring</td>
<td>699</td>
</tr>
<tr>
<td>zero?</td>
<td>124</td>
</tr>
<tr>
<td>zip</td>
<td>594</td>
</tr>
<tr>
<td>zlib-version</td>
<td>958</td>
</tr>
<tr>
<td>zstream-adler32</td>
<td>958</td>
</tr>
<tr>
<td>zstream-data-type</td>
<td>958</td>
</tr>
<tr>
<td>zstream-dictionary-adler32</td>
<td>958</td>
</tr>
<tr>
<td>zstream-params-set!</td>
<td>958</td>
</tr>
<tr>
<td>zstream-total-in</td>
<td>958</td>
</tr>
<tr>
<td>zstream-total-out</td>
<td>958</td>
</tr>
</tbody>
</table>
## Appendix C Module Index

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binary.io</code></td>
<td>808</td>
</tr>
<tr>
<td><code>binary.pack</code></td>
<td>811</td>
</tr>
<tr>
<td><code>compat.chibi-test</code></td>
<td>814</td>
</tr>
<tr>
<td><code>compat.norational</code></td>
<td>814</td>
</tr>
<tr>
<td><code>compat.real-elementary-functions</code></td>
<td>815</td>
</tr>
<tr>
<td><code>control.cseq</code></td>
<td>816</td>
</tr>
<tr>
<td><code>control.future</code></td>
<td>816</td>
</tr>
<tr>
<td><code>control.job</code></td>
<td>817</td>
</tr>
<tr>
<td><code>control.plumbing</code></td>
<td>819</td>
</tr>
<tr>
<td><code>control.pmap</code></td>
<td>822</td>
</tr>
<tr>
<td><code>control.scheduler</code></td>
<td>824</td>
</tr>
<tr>
<td><code>control.thread-pool</code></td>
<td>826</td>
</tr>
<tr>
<td><code>crypt.bcrypt</code></td>
<td>827</td>
</tr>
<tr>
<td><code>data.cache</code></td>
<td>828</td>
</tr>
<tr>
<td><code>data.heap</code></td>
<td>831</td>
</tr>
<tr>
<td><code>data.idenque</code></td>
<td>833</td>
</tr>
<tr>
<td><code>data.imap</code></td>
<td>834</td>
</tr>
<tr>
<td><code>data.priority-map</code></td>
<td>836</td>
</tr>
<tr>
<td><code>data.queue</code></td>
<td>837</td>
</tr>
<tr>
<td><code>data.random</code></td>
<td>841</td>
</tr>
<tr>
<td><code>data.range</code></td>
<td>845</td>
</tr>
<tr>
<td><code>data.skew-list</code></td>
<td>849</td>
</tr>
<tr>
<td><code>data.skelist</code></td>
<td>851</td>
</tr>
<tr>
<td><code>data.sparse</code></td>
<td>853</td>
</tr>
<tr>
<td><code>data.trie</code></td>
<td>859</td>
</tr>
<tr>
<td><code>data.udid</code></td>
<td>863, 864</td>
</tr>
<tr>
<td><code>dbi</code></td>
<td>865</td>
</tr>
<tr>
<td><code>dbm</code></td>
<td>871</td>
</tr>
<tr>
<td><code>dbm.fsdbm</code></td>
<td>876</td>
</tr>
<tr>
<td><code>dbm.gdbm</code></td>
<td>876</td>
</tr>
<tr>
<td><code>dbm.ndbm</code></td>
<td>878</td>
</tr>
<tr>
<td><code>dbm.odbm</code></td>
<td>879</td>
</tr>
<tr>
<td><code>file.filter</code></td>
<td>880</td>
</tr>
<tr>
<td><code>file.util</code></td>
<td>881</td>
</tr>
<tr>
<td><code>gauche</code></td>
<td>86</td>
</tr>
<tr>
<td><code>gauche.array</code></td>
<td>366</td>
</tr>
<tr>
<td><code>gauche.base</code></td>
<td>373</td>
</tr>
<tr>
<td><code>gauche.bitvector</code></td>
<td>374</td>
</tr>
<tr>
<td><code>gauche.cgen</code></td>
<td>383</td>
</tr>
<tr>
<td><code>gauche.charconv</code></td>
<td>399</td>
</tr>
<tr>
<td><code>gauche.collection</code></td>
<td>404</td>
</tr>
<tr>
<td><code>gauche.config</code></td>
<td>413</td>
</tr>
<tr>
<td><code>gauche.config</code></td>
<td>413</td>
</tr>
<tr>
<td><code>gauche.connection</code></td>
<td>426</td>
</tr>
<tr>
<td><code>gauche.dictionary</code></td>
<td>428</td>
</tr>
<tr>
<td><code>gauche.fcntl</code></td>
<td>432</td>
</tr>
<tr>
<td><code>gauche.generator</code></td>
<td>436</td>
</tr>
<tr>
<td><code>gauche.hook</code></td>
<td>447</td>
</tr>
<tr>
<td><code>gauche.interactive</code></td>
<td>448</td>
</tr>
<tr>
<td><code>gauche.keyword</code></td>
<td>86</td>
</tr>
<tr>
<td><code>gauche.lazy</code></td>
<td>451</td>
</tr>
<tr>
<td><code>gauche.listener</code></td>
<td>455</td>
</tr>
<tr>
<td><code>gauche.logger</code></td>
<td>458</td>
</tr>
<tr>
<td><code>gauche.mop.instance-pool</code></td>
<td>460</td>
</tr>
<tr>
<td><code>gauche.mop.propagate</code></td>
<td>461</td>
</tr>
<tr>
<td><code>gauche.mop.singleton</code></td>
<td>463</td>
</tr>
<tr>
<td><code>gauche.mop.validator</code></td>
<td>463</td>
</tr>
<tr>
<td><code>gauche.net</code></td>
<td>464</td>
</tr>
<tr>
<td><code>gauche.package</code></td>
<td>477</td>
</tr>
<tr>
<td><code>gauche.parameter</code></td>
<td>480</td>
</tr>
<tr>
<td><code>gauche.parseopt</code></td>
<td>480</td>
</tr>
<tr>
<td><code>gauche.partcont</code></td>
<td>485</td>
</tr>
<tr>
<td><code>gauche.process</code></td>
<td>488</td>
</tr>
<tr>
<td><code>gauche.record</code></td>
<td>502</td>
</tr>
<tr>
<td><code>gauche.reload</code></td>
<td>508</td>
</tr>
<tr>
<td><code>gauche.selector</code></td>
<td>509</td>
</tr>
<tr>
<td><code>gauche.sequence</code></td>
<td>510</td>
</tr>
<tr>
<td><code>gauche.syslog</code></td>
<td>517</td>
</tr>
<tr>
<td><code>gauche.termios</code></td>
<td>518</td>
</tr>
<tr>
<td><code>gauche.test</code></td>
<td>521</td>
</tr>
<tr>
<td><code>gauche.threads</code></td>
<td>530</td>
</tr>
<tr>
<td><code>gauche.time</code></td>
<td>546</td>
</tr>
<tr>
<td><code>gauche.unicode</code></td>
<td>549</td>
</tr>
<tr>
<td><code>gauche.uvector</code></td>
<td>556</td>
</tr>
<tr>
<td><code>gauche.version</code></td>
<td>570</td>
</tr>
<tr>
<td><code>gauche.version</code></td>
<td>572</td>
</tr>
<tr>
<td><code>keyword</code></td>
<td>86</td>
</tr>
<tr>
<td><code>math.const</code></td>
<td>893</td>
</tr>
<tr>
<td><code>math.mt-random</code></td>
<td>893</td>
</tr>
<tr>
<td><code>math.prime</code></td>
<td>895</td>
</tr>
<tr>
<td><code>math.simplex</code></td>
<td>897</td>
</tr>
<tr>
<td><code>null</code></td>
<td>85</td>
</tr>
<tr>
<td><code>os.windows</code></td>
<td>898</td>
</tr>
<tr>
<td><code>parser.peg</code></td>
<td>902</td>
</tr>
</tbody>
</table>
Appendix C: Module Index

R

rfc.822 ........................................ 917
rfc.base64 .................................... 921
rfc.cookie .................................... 924
rfc.ftp ....................................... 925
rfc.hmac .................................... 928
rfc.http ..................................... 930
rfc.icmp ..................................... 935
rfc.ip ......................................... 936
rfc.json ....................................... 937
rfc.md5 ....................................... 939
rfc.mime ..................................... 939
rfc.quoted-printable ....................... 945
rfc.sha ....................................... 946
rfc.shal ...................................... 946
rfc.tls ....................................... 946
rfc.uri ....................................... 949
rfc.uuid ...................................... 954
rfc.zlib ..................................... 955

S

scheme ........................................ 85
scheme.base ................................ 585
scheme.bitwise .............................. 664
scheme.box .................................. 636
scheme.bytevector ......................... 679
scheme.case-lambda ......................... 587
scheme.char .................................. 587
scheme.charset .............................. 614
scheme.comparator ......................... 640
scheme.complex .............................. 588
scheme.cxr .................................. 588
scheme.division ............................. 663
scheme.ephemeron ........................... 639
scheme.eval .................................. 589
scheme.file .................................. 589
scheme.finux .................................. 668
scheme.finux .................................. 670
scheme.finux .................................. 670
scheme.generator ........................... 631
scheme.hash .................................. 618
scheme.hash-table ........................... 618
scheme.ideque ................................ 623
scheme.ilex .................................. 621
scheme.inexact ............................... 589
scheme.lazy .................................. 590
scheme.list .................................. 593
scheme.list-queue ......................... 636
scheme.load .................................. 590
scheme.lesq .................................. 633
scheme.mapping ............................... 652
scheme.mapping.hash ....................... 652
scheme.process-context ..................... 590
scheme.r5rs .................................. 592
scheme.read ................................... 591
scheme.regex .................................. 640
scheme.repl .................................. 591
scheme.rlist .................................. 622
scheme.set .................................... 606
scheme.show .................................. 681
scheme.sort .................................. 602
scheme.stream ............................... 635
scheme.text .................................. 627
scheme.time .................................. 684
scheme.vector ................................ 597

scheme.vector.base ......................... 602
scheme.vector.c128 ......................... 602
scheme.vector.c64 ......................... 602
scheme.vector.f32 ......................... 602
scheme.vector.f64 ......................... 602
scheme.vector.s16 ......................... 602
scheme.vector.s32 ......................... 602
scheme.vector.s64 ......................... 602
scheme.vector.s8 ......................... 602
scheme.vector.u16 ......................... 602
scheme.vector.u32 ......................... 602
scheme.vector.u64 ......................... 602
scheme.vector.u8 ......................... 602
scheme.write .................................. 592
slib .......................................... 959
srfi.101 ..................................... 727
srfi.106 ..................................... 727
srfi.112 ..................................... 730
srfi.114 ..................................... 731
srfi.118 ..................................... 736
srfi.120 ..................................... 736
srfi.129 ..................................... 738
srfi.13 ........................................ 692
srfi.130 ..................................... 738
srfi.152 ..................................... 740
srfi.154 ..................................... 742
srfi.160 ..................................... 743
srfi.162 ..................................... 744
srfi.170 ..................................... 744
srfi.172 ..................................... 750
srfi.172.functional ........................ 750
srfi.173 ..................................... 751
srfi.174 ..................................... 752
srfi.175 ..................................... 753
srfi.176 ..................................... 756
srfi.180 ..................................... 757
srfi.181 ..................................... 760
srfi.185 ..................................... 764
srfi.189 ..................................... 767
srfi.19 ........................................ 701
srfi.192 ..................................... 773
srfi.193 ..................................... 773
srfi.196 ..................................... 774
srfi.197 ..................................... 774
srfi.207 ..................................... 776
srfi.209 ..................................... 779
srfi.210 ..................................... 784
srfi.216 ..................................... 786
srfi.217 ..................................... 787
srfi.219 ..................................... 792
srfi.221 ..................................... 792
srfi.222 ..................................... 794
srfi.226 ..................................... 796
srfi.227 ..................................... 800
srfi.227.definitions ......................... 800
srfi.228 ..................................... 801
srfi.229 ..................................... 802
srfi.232 ..................................... 803
srfi.235 ..................................... 804
srfi.239 ..................................... 807
srfi.27 ........................................ 706
srfi.29 ........................................ 707
srfi.29.bundle ............................... 707
srfi.29.format ............................... 707
srfi.37 ........................................ 709
Appendix C: Module Index

srfi.4 ........................................ 691
srfi.42 ........................................ 711
srfi.43 ........................................ 717
srfi.5 ........................................ 691
srfi.55 ....................................... 718
srfi.60 ....................................... 718
srfi.64 ....................................... 720
srfi.66 ....................................... 720
srfi.69 ....................................... 721
srfi.7 .......................................... 691
srfi.74 ....................................... 723
srfi.78 ....................................... 725
srfi.98 ....................................... 727
sxml.serializer ............................. 983
sxml.ssax .................................. 959
sxml.sxpath ................................ 970
sxml.tools .................................. 978

T
text.console .................................. 985
text.csv ...................................... 988
text.diff ...................................... 991
text.edn ...................................... 994
text.external-editor ......................... 997
text.gap-buffer .............................. 998
text.gettext ................................ 1000
text.html-lite ................................ 1002
text.info ..................................... 1004
text.multicolumn ............................ 1006
text.pager ................................... 1006
text.parse ................................... 1007
text.progress ................................ 1008
text.segmented-match ....................... 1010
text.sql ....................................... 1011
text.template ................................. 1012
text.tr ......................................... 1013
text.tree ...................................... 1014
text.unicode ................................ 364

U
user ............................................. 86
util.combinations .......................... 1015
util.digest ................................... 1017
util.dominator .............................. 1018
util.isomorph ................................ 1020
util.lcs ....................................... 1020
util.levenshtein ............................ 1023
util.list ...................................... 364
util.match .................................... 1024
util.queue .................................... 364
util.rbtree ................................... 1024
util.record ................................. 1029
util.relation ................................. 1031
util.sparse ................................... 364
util.stream ................................... 1033
util.temporal-relation ..................... 1043
util.toposort ............................... 1045
util.trie ..................................... 364
util.unification ............................ 1046

W
www.cgi ....................................... 1047
www.cgi.test ................................ 1052
Appendix D  Lexical syntax index

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>,</td>
</tr>
<tr>
<td>#!</td>
<td>46</td>
</tr>
<tr>
<td>&quot;</td>
<td>173</td>
</tr>
<tr>
<td>#*</td>
<td>203</td>
</tr>
<tr>
<td>#**</td>
<td>182</td>
</tr>
<tr>
<td>#,</td>
<td>273</td>
</tr>
<tr>
<td>#/</td>
<td>183</td>
</tr>
<tr>
<td>#=&gt;</td>
<td>165</td>
</tr>
<tr>
<td>#\</td>
<td>174</td>
</tr>
<tr>
<td>#c128</td>
<td>199</td>
</tr>
<tr>
<td>#c32</td>
<td>199</td>
</tr>
<tr>
<td>#c64</td>
<td>199</td>
</tr>
<tr>
<td>#f16</td>
<td>199</td>
</tr>
<tr>
<td>#f32</td>
<td>199</td>
</tr>
<tr>
<td>#f64</td>
<td>199</td>
</tr>
<tr>
<td>#s16</td>
<td>199</td>
</tr>
<tr>
<td>#s32</td>
<td>199</td>
</tr>
<tr>
<td>#s64</td>
<td>199</td>
</tr>
<tr>
<td>#s8</td>
<td>199</td>
</tr>
<tr>
<td>#u16</td>
<td>199</td>
</tr>
<tr>
<td>#u32</td>
<td>199</td>
</tr>
<tr>
<td>#u64</td>
<td>199</td>
</tr>
<tr>
<td>#u8</td>
<td>199</td>
</tr>
</tbody>
</table>
## Appendix E Class Index

For readability, the surrounding `< and >` are stripped off.

<table>
<thead>
<tr>
<th>Class Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>108</td>
</tr>
<tr>
<td>/</td>
<td>107</td>
</tr>
<tr>
<td>^</td>
<td>108</td>
</tr>
<tr>
<td>abandoned-mutex-exception</td>
<td>544</td>
</tr>
<tr>
<td>array</td>
<td>366</td>
</tr>
<tr>
<td>array-base</td>
<td>366</td>
</tr>
<tr>
<td>ax-tls</td>
<td>948</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>barrier</td>
<td>542</td>
</tr>
<tr>
<td>bimap</td>
<td>430</td>
</tr>
<tr>
<td>binary-heap</td>
<td>872</td>
</tr>
<tr>
<td>bitvector</td>
<td>203</td>
</tr>
<tr>
<td>boolean</td>
<td>139</td>
</tr>
<tr>
<td>bottom</td>
<td>106</td>
</tr>
<tr>
<td>buffered-input-port</td>
<td>575</td>
</tr>
<tr>
<td>buffered-output-port</td>
<td>575</td>
</tr>
<tr>
<td>c128vector</td>
<td>199</td>
</tr>
<tr>
<td>c32vector</td>
<td>199</td>
</tr>
<tr>
<td>c64vector</td>
<td>199</td>
</tr>
<tr>
<td>cgen-node</td>
<td>386</td>
</tr>
<tr>
<td>cgen-type</td>
<td>390</td>
</tr>
<tr>
<td>cgen-unit</td>
<td>383</td>
</tr>
<tr>
<td>char</td>
<td>160</td>
</tr>
<tr>
<td>char-set</td>
<td>165</td>
</tr>
<tr>
<td>class</td>
<td>352</td>
</tr>
<tr>
<td>complex</td>
<td>123</td>
</tr>
<tr>
<td>compound-condition</td>
<td>254</td>
</tr>
<tr>
<td>concurrent-modification-violation</td>
<td>544</td>
</tr>
<tr>
<td>condition</td>
<td>254</td>
</tr>
<tr>
<td>condition-meta</td>
<td>254</td>
</tr>
<tr>
<td>condition-variable</td>
<td>539</td>
</tr>
<tr>
<td>continuation-violation</td>
<td>256</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>703</td>
</tr>
<tr>
<td>dbi-connection</td>
<td>866</td>
</tr>
<tr>
<td>dbi-driver</td>
<td>867</td>
</tr>
<tr>
<td>dbi-query</td>
<td>867</td>
</tr>
<tr>
<td>dbm</td>
<td>872</td>
</tr>
<tr>
<td>dbm-meta</td>
<td>873</td>
</tr>
<tr>
<td>deflating-port</td>
<td>956</td>
</tr>
<tr>
<td>double</td>
<td>109</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>edn-object</td>
<td>995</td>
</tr>
<tr>
<td>error</td>
<td>254</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>f16array</td>
<td>366</td>
</tr>
<tr>
<td>f16vector</td>
<td>199</td>
</tr>
<tr>
<td>f32array</td>
<td>366</td>
</tr>
<tr>
<td>f32vector</td>
<td>199</td>
</tr>
<tr>
<td>f64array</td>
<td>366</td>
</tr>
<tr>
<td>f64vector</td>
<td>199</td>
</tr>
<tr>
<td>fixnum</td>
<td>109</td>
</tr>
<tr>
<td>float</td>
<td>109</td>
</tr>
<tr>
<td>fsdbm</td>
<td>876</td>
</tr>
<tr>
<td>ftp-connection</td>
<td>925</td>
</tr>
<tr>
<td>ftp-error</td>
<td>926</td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>gdbm</td>
<td>876</td>
</tr>
<tr>
<td>H</td>
<td></td>
</tr>
<tr>
<td>hash-table</td>
<td>207</td>
</tr>
<tr>
<td>hmac</td>
<td>929</td>
</tr>
<tr>
<td>hook</td>
<td>448</td>
</tr>
<tr>
<td>http-error</td>
<td>930</td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>identifier</td>
<td>98</td>
</tr>
<tr>
<td>inflating-port</td>
<td>956</td>
</tr>
<tr>
<td>info-file</td>
<td>1004</td>
</tr>
<tr>
<td>info-node</td>
<td>1004</td>
</tr>
<tr>
<td>instance-pool-meta</td>
<td>461</td>
</tr>
<tr>
<td>instance-pool-mixin</td>
<td>461</td>
</tr>
<tr>
<td>int</td>
<td>109</td>
</tr>
<tr>
<td>int16</td>
<td>109</td>
</tr>
<tr>
<td>int32</td>
<td>109</td>
</tr>
<tr>
<td>int64</td>
<td>109</td>
</tr>
<tr>
<td>int8</td>
<td>109</td>
</tr>
<tr>
<td>integer</td>
<td>123</td>
</tr>
<tr>
<td>io-closed-error</td>
<td>256</td>
</tr>
<tr>
<td>io-error</td>
<td>256</td>
</tr>
<tr>
<td>io-read-error</td>
<td>256</td>
</tr>
<tr>
<td>io-unit-error</td>
<td>256</td>
</tr>
<tr>
<td>io-write-error</td>
<td>256</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>join-timeout-exception</td>
<td>544</td>
</tr>
<tr>
<td>K</td>
<td></td>
</tr>
<tr>
<td>keyword</td>
<td>157</td>
</tr>
</tbody>
</table>
### Appendix E: Class Index

<table>
<thead>
<tr>
<th>L</th>
<th>1125</th>
</tr>
</thead>
<tbody>
<tr>
<td>latch</td>
<td>542</td>
</tr>
<tr>
<td>List</td>
<td>108</td>
</tr>
<tr>
<td>list</td>
<td>141</td>
</tr>
<tr>
<td>listener</td>
<td>455</td>
</tr>
<tr>
<td>log-drain</td>
<td>458</td>
</tr>
<tr>
<td>long</td>
<td>109</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping</td>
<td>653</td>
</tr>
<tr>
<td>mbedtls</td>
<td>948</td>
</tr>
<tr>
<td>md5</td>
<td>939</td>
</tr>
<tr>
<td>mersenne-twister</td>
<td>893</td>
</tr>
<tr>
<td>message-condition</td>
<td>254</td>
</tr>
<tr>
<td>message-digest-algorithm</td>
<td>1018</td>
</tr>
<tr>
<td>message-digest-algorithm-meta</td>
<td>1018</td>
</tr>
<tr>
<td>mime-message</td>
<td>942</td>
</tr>
<tr>
<td>module</td>
<td>84</td>
</tr>
<tr>
<td>mtqueue</td>
<td>837</td>
</tr>
<tr>
<td>mutex</td>
<td>536</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ndbm</td>
<td>878</td>
</tr>
<tr>
<td>null</td>
<td>141</td>
</tr>
<tr>
<td>number</td>
<td>123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>106</td>
</tr>
<tr>
<td>object-set-relation</td>
<td>1033</td>
</tr>
<tr>
<td>odbm</td>
<td>879</td>
</tr>
<tr>
<td>off_t</td>
<td>109</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pair</td>
<td>141</td>
</tr>
<tr>
<td>port</td>
<td>259</td>
</tr>
<tr>
<td>port-error</td>
<td>256</td>
</tr>
<tr>
<td>procedure</td>
<td>217</td>
</tr>
<tr>
<td>process</td>
<td>494</td>
</tr>
<tr>
<td>process-abnormal-exit</td>
<td>495</td>
</tr>
<tr>
<td>process-time-counter</td>
<td>548</td>
</tr>
<tr>
<td>propagate-meta</td>
<td>462</td>
</tr>
<tr>
<td>propagate-mixin</td>
<td>463</td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td>109</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>queue</td>
<td>837</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>846</td>
</tr>
<tr>
<td>range-meta</td>
<td>846</td>
</tr>
<tr>
<td>rational</td>
<td>123</td>
</tr>
<tr>
<td>rbtree</td>
<td>364</td>
</tr>
<tr>
<td>read-error</td>
<td>255</td>
</tr>
<tr>
<td>real</td>
<td>123</td>
</tr>
<tr>
<td>real-time-counter</td>
<td>548</td>
</tr>
<tr>
<td>regexp</td>
<td>186</td>
</tr>
<tr>
<td>regmatch</td>
<td>186</td>
</tr>
<tr>
<td>relation</td>
<td>1031</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s16array</td>
<td>366</td>
</tr>
<tr>
<td>s16vector</td>
<td>199</td>
</tr>
<tr>
<td>s32array</td>
<td>366</td>
</tr>
<tr>
<td>s32vector</td>
<td>199</td>
</tr>
<tr>
<td>s64array</td>
<td>366</td>
</tr>
<tr>
<td>s64vector</td>
<td>199</td>
</tr>
<tr>
<td>s8array</td>
<td>366</td>
</tr>
<tr>
<td>s8vector</td>
<td>199</td>
</tr>
<tr>
<td>scheduler</td>
<td>824</td>
</tr>
<tr>
<td>selector</td>
<td>509</td>
</tr>
<tr>
<td>semaphore</td>
<td>541</td>
</tr>
<tr>
<td>serious-condition</td>
<td>254</td>
</tr>
<tr>
<td>serious-condition</td>
<td>254</td>
</tr>
<tr>
<td>sha1</td>
<td>946</td>
</tr>
<tr>
<td>sha224</td>
<td>946</td>
</tr>
<tr>
<td>sha256</td>
<td>946</td>
</tr>
<tr>
<td>sha384</td>
<td>946</td>
</tr>
<tr>
<td>sha512</td>
<td>946</td>
</tr>
<tr>
<td>short</td>
<td>109</td>
</tr>
<tr>
<td>simple-Relation</td>
<td>1033</td>
</tr>
<tr>
<td>singleton-meta</td>
<td>463</td>
</tr>
<tr>
<td>singleton-mixin</td>
<td>463</td>
</tr>
<tr>
<td>size_t</td>
<td>109</td>
</tr>
<tr>
<td>skew-list</td>
<td>852</td>
</tr>
<tr>
<td>sockaddr</td>
<td>465</td>
</tr>
<tr>
<td>sockaddr-in</td>
<td>465</td>
</tr>
<tr>
<td>sockaddr-in6</td>
<td>466</td>
</tr>
<tr>
<td>sockaddr-un</td>
<td>466</td>
</tr>
<tr>
<td>socket</td>
<td>467</td>
</tr>
<tr>
<td>sparse-f16matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-f16vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-f32matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-f32vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-f64matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-f64vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-matrix-base</td>
<td>857</td>
</tr>
<tr>
<td>sparse-s16matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-s16vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-s32matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-s32vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-s64matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-s64vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-s8matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-s8vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-table</td>
<td>858</td>
</tr>
<tr>
<td>sparse-u16matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-u16vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-u32matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-u32vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-u64matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-u64vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-u8matrix</td>
<td>857</td>
</tr>
<tr>
<td>sparse-u8vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-vector</td>
<td>855</td>
</tr>
<tr>
<td>sparse-vector-base</td>
<td>854</td>
</tr>
<tr>
<td>ssize_t</td>
<td>109</td>
</tr>
<tr>
<td>string</td>
<td>170</td>
</tr>
<tr>
<td>symbol</td>
<td>155</td>
</tr>
<tr>
<td>sys-addrinfo</td>
<td>477</td>
</tr>
<tr>
<td>sys-fdset</td>
<td>322</td>
</tr>
<tr>
<td>sys-flock</td>
<td>434</td>
</tr>
<tr>
<td>sys-group</td>
<td>304</td>
</tr>
<tr>
<td>Category</td>
<td>Class Name</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>sys</td>
<td>sys-hostent</td>
</tr>
<tr>
<td></td>
<td>sys-passwd</td>
</tr>
<tr>
<td></td>
<td>sys-protoent</td>
</tr>
<tr>
<td></td>
<td>sys-servent</td>
</tr>
<tr>
<td></td>
<td>sys-sigset</td>
</tr>
<tr>
<td></td>
<td>sys-stat</td>
</tr>
<tr>
<td></td>
<td>sys-statvfs</td>
</tr>
<tr>
<td></td>
<td>sys-termios</td>
</tr>
<tr>
<td></td>
<td>sys-tm</td>
</tr>
<tr>
<td></td>
<td>system-error</td>
</tr>
<tr>
<td></td>
<td>system-time-counter</td>
</tr>
<tr>
<td>T</td>
<td>terminated-thread-exception</td>
</tr>
<tr>
<td></td>
<td>thread</td>
</tr>
<tr>
<td></td>
<td>thread-exception</td>
</tr>
<tr>
<td></td>
<td>thread-pool</td>
</tr>
<tr>
<td></td>
<td>time</td>
</tr>
<tr>
<td></td>
<td>time-counter</td>
</tr>
<tr>
<td></td>
<td>tls</td>
</tr>
<tr>
<td></td>
<td>top</td>
</tr>
<tr>
<td></td>
<td>tree-map</td>
</tr>
<tr>
<td></td>
<td>trie</td>
</tr>
<tr>
<td></td>
<td>Tuple</td>
</tr>
<tr>
<td>U</td>
<td>u16array</td>
</tr>
<tr>
<td></td>
<td>u16vector</td>
</tr>
<tr>
<td></td>
<td>u32array</td>
</tr>
<tr>
<td></td>
<td>u32vector</td>
</tr>
<tr>
<td></td>
<td>u64array</td>
</tr>
<tr>
<td></td>
<td>u64vector</td>
</tr>
<tr>
<td></td>
<td>u8array</td>
</tr>
<tr>
<td></td>
<td>u8vector</td>
</tr>
<tr>
<td></td>
<td>uint</td>
</tr>
<tr>
<td></td>
<td>uint16</td>
</tr>
<tr>
<td></td>
<td>uint32</td>
</tr>
<tr>
<td></td>
<td>uint64</td>
</tr>
<tr>
<td></td>
<td>uint8</td>
</tr>
<tr>
<td></td>
<td>ulong</td>
</tr>
<tr>
<td></td>
<td>uncaught-exception</td>
</tr>
<tr>
<td></td>
<td>unhandled-signal-error</td>
</tr>
<tr>
<td></td>
<td>user-time-counter</td>
</tr>
<tr>
<td></td>
<td>ushman</td>
</tr>
<tr>
<td>V</td>
<td>validator-meta</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
</tr>
<tr>
<td></td>
<td>vector</td>
</tr>
<tr>
<td></td>
<td>virtual-input-port</td>
</tr>
<tr>
<td></td>
<td>virtual-output-port</td>
</tr>
<tr>
<td></td>
<td>void</td>
</tr>
<tr>
<td></td>
<td>vt100</td>
</tr>
<tr>
<td>W</td>
<td>weak-vector</td>
</tr>
<tr>
<td></td>
<td>win:console-screen-buffer-info</td>
</tr>
<tr>
<td></td>
<td>win:input-record</td>
</tr>
<tr>
<td></td>
<td>windows-console</td>
</tr>
<tr>
<td></td>
<td>write-controls</td>
</tr>
</tbody>
</table>
Appendix F Variable Index

&
&concurrent-modification .................................... 544
&condition .................................................. 254
&continuation ................................................ 256
&error .......................................................... 254
&io-closed-error ........................................... 256
&io-error ..................................................... 256
&io-port-error ............................................... 256
&io-read-error ............................................... 256
&io-write-error ............................................. 256
&read-error .................................................. 255
&serious ..................................................... 254
&thread ....................................................... 543
&thread-abandoned-mutex ................................... 544
&thread-already-terminated ................................ 544
&thread-timeout ............................................ 544
&uncaught-exception ....................................... 544

* 
*af-inet* .................................................... 729
*af-inet6* .................................................. 729
*af-unspec* ................................................ 729
*ai-addrconfig* ......................................... 729
*ai-all* ..................................................... 729
*ai-canonname* ............................................ 729
*ai-numericname* .......................................... 729
*ai-v4mapped* ............................................ 729
*argv* ........................................................ 294
*ippproto-ip* ............................................. 729
*ippproto-udp* ............................................ 729
*load-path* ................................................. 286
*msg-none* ................................................ 730
*msg-oob* .................................................. 730
*msg-peek* ................................................ 730
*msg-waitall* ............................................... 730
*primes* .................................................... 895
*program-name* ............................................. 294
/rfc2396-unreserved-char-set* ......................... 954
/rfc3986-unreserved-char-set* ......................... 954
/rfc822-axext-chars* ................................... 919
/rfc822-standard-tokenizers* ........................... 919
*shut-rd* ................................................... 730
*shut-rdw* ................................................. 730
*shut-ur* ................................................... 730
*small-prime-bound* ...................................... 895
*sock-dgram* ............................................... 729
*sock-stream* .............................................. 729
*test-error* ............................................... 526
*test-report-error* ...................................... 526

\pi..................................................... 893
\pi/180 ................................................... 893
\pi/2 .................................................... 893
\pi/4 .................................................... 893
1 .......................... 893
1/\pi .................................................... 893
1/pi ..................................................... 893
180/\pi .................................................. 893
180/pi .................................................. 893
2
2\pi ..................................................... 893
2pi ....................................................... 893

A
accessors of <class> ....................................... 353
addr of <sys-addrinfo> .................................. 477
address of <memory-region> .............................. 323
addresses of <sys-hostent> .............................. 475
addrlen of <sys-addrinfo> ............................... 477
AF_INET .................................................. 470
AF_INET6 ............................................... 470
AF_UNIX ................................................ 470
aliases of <sys-hostent> .................................. 475
daliases of <sys-servent> ................................ 476
atim of <sys-stat> ....................................... 302
atime of <sys-stat> ...................................... 302
attributes of <win:console-screen-buffer-info> ..... 900
authors of <gauche-package-description> ............ 479

B
BACKGROUND_BLUE ......................................... 900
BACKGROUND_GREEN ........................................ 900
BACKGROUND_INTENSITY .................................... 900
BACKGROUND_RED .......................................... 900
bag-comparator ........................................... 614
base of <write-controls> ................................ 276
bavall of <sys-statvfs> .................................. 435
bfree of <sys-statvfs> .................................... 435
binary-input ............................................... 745
binary-input/output ....................................... 745
binary-output ............................................. 745
blocks of <sys-statvfs> .................................. 435
boolean-comparator ....................................... 120
bsize of <gdbm> ........................................... 876
bsize of <sys-statvfs> .................................... 435
buffer-block ............................................. 745
buffer-line .............................................. 745
buffer-none ............................................. 745
bytevector-comparator .................................. 121

©
@vector-comparator ....................................... 562
Appendix F: Variable Index

C

c-file of <gen-unit> ............................................. 383
c128vector-comparator ........................................ 562
c32vector-comparator ........................................ 562
c64vector-comparator ........................................ 562
category of <class> ........................................... 353
cc of <sys-termi> ............................................ 519
cflag of <sys-termi> ............................................ 519
cchar-ci-comparator ........................................... 120
cchar-comparator .............................................. 120
cchar-set:ascii ................................................ 168
cchar-set:ascii-blank ........................................ 168
cchar-set:ascii-control ...................................... 168
cchar-set:ascii-digit ......................................... 168
cchar-set:ascii-graphic ...................................... 168
cchar-set:ascii-letter ....................................... 168
cchar-set:ascii-letter-digit ................................ 168
cchar-set:ascii-lower-case ................................ 168
cchar-set:ascii-printing .................................... 168
cchar-set:ascii-punctuation ................................ 168
cchart:ascii-symbol ......................................... 168
cchar-set:ascii-upper-case ................................ 168
cchar-set:ascii-whitespace ................................ 168
cchar-set:ascii-word ........................................ 168
cchar-set:blank ............................................... 167
cchar-set:C ...................................................... 169
cchar-set:Cc ..................................................... 169
cchar-set: Cf .................................................... 169
cchar-set:Cn ..................................................... 169
cchar-set:Co ..................................................... 169
cchar-set:Cs ...................................................... 169
cchar-set:Digit ................................................ 167
cchar-set:empty ................................................. 168
cchar-set:full .................................................. 168
cchar-set:graphic .............................................. 167
cchar-set:hex-digit .......................................... 167
cchar-set:iso-control ........................................ 168
cchar-set:letter ............................................... 167
cchar-set:letter+digit ....................................... 167
cchar-set:lower-case ........................................ 167
cchar-set:lower-case ........................................ 167
cchar-set:OS ..................................................... 169
cchar-set:So ..................................................... 169
cchar-set:Sk ..................................................... 169
cchar-set:S ....................................................... 169
cchar-set:S ....................................................... 169
cchar-set:S ....................................................... 169
cchar-set:S ....................................................... 169
class of <sys-passwd> ....................................... 305
close of <buffered-input-port> ................................ 575
close of <buffered-output-port> ................................ 576
close of <virtual-input-port> ................................. 573
close of <virtual-output-port> ................................. 574
closed of <queue> .............................................. 837
col .......................................................... 687
column of <read-error> ....................................... 255
ccomma-rule .................................................... 688
ccomma-sep ..................................................... 688
comparator-one ............................................... 802
comparator-zero ............................................... 802
complex-comparator .......................................... 121
component of <gauche-package-description> .......... 479
cconnection of <dbi-query> .................................. 868
ccontent of <info-node> ...................................... 1004
ccontent of <mime-part> ...................................... 943
count of <time-result> ....................................... 547
cpl of <class> .................................................. 353
cctime of <sys-stat> .......................................... 302
CTRL_BREAKEVENT ............................................ 899
CTRL_CEVENT .................................................. 899
cursor-position.x of ...
<win:console-screen-buffer-info> .................. 900
cursor-position.y of ...
<win:console-screen-buffer-info> .................. 900

day of <date> .................................................. 703
decimal-align .................................................. 688
decimal-sep ..................................................... 688
default-comparator ............................................ 119
default-random-source ...................................... 707
defined-modules of <class> ................................. 353
description of ...
<gauche-package-description> ....................... 478
dev of <sys-stat> ............................................. 302
dir of <sys-passwd> .......................................... 305
direct-methods of <class> ................................. 353
direct-slots of <class> ...................................... 353
direct-subclasses of <class> ......................... 355
direct-supers of <class> .................................. 353
driver-name of ...
<dbi-nonexistent-driver-error> .................. 866
## Appendix F: Variable Index

### E

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>893</td>
</tr>
<tr>
<td>edn-comparator</td>
<td>995</td>
</tr>
<tr>
<td>ellipse</td>
<td>687</td>
</tr>
<tr>
<td>ENABLE_ECHO_INPUT</td>
<td>899</td>
</tr>
<tr>
<td>ENABLE_LINE_INPUT</td>
<td>899</td>
</tr>
<tr>
<td>ENABLE_MOUSE_INPUT</td>
<td>899</td>
</tr>
<tr>
<td>ENABLE_PROCESSED_INPUT</td>
<td>899</td>
</tr>
<tr>
<td>ENABLE_PROCESSED_OUTPUT</td>
<td>899</td>
</tr>
<tr>
<td>ENABLE_WINDOW_INPUT</td>
<td>899</td>
</tr>
<tr>
<td>ENABLE_WRAP_AT_EOL_OUTPUT</td>
<td>899</td>
</tr>
<tr>
<td>environment of &lt;listener&gt;</td>
<td>456</td>
</tr>
<tr>
<td>eq-comparator</td>
<td>456</td>
</tr>
<tr>
<td>equal-comparator</td>
<td>120</td>
</tr>
<tr>
<td>e &lt;system-error&gt;</td>
<td>255</td>
</tr>
<tr>
<td>error-handler of &lt;listener&gt;</td>
<td>456</td>
</tr>
<tr>
<td>error-handler of &lt;scheduler&gt;</td>
<td>825</td>
</tr>
<tr>
<td>error-port of &lt;listener&gt;</td>
<td>456</td>
</tr>
<tr>
<td>evaluator of &lt;listener&gt;</td>
<td>456</td>
</tr>
<tr>
<td>event-type of <a href="">win:input-record</a></td>
<td>900</td>
</tr>
<tr>
<td>exact-integer-comparator</td>
<td>121</td>
</tr>
</tbody>
</table>

### F

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>f16vector-comparator</td>
<td>562</td>
</tr>
<tr>
<td>f32vector-comparator</td>
<td>562</td>
</tr>
<tr>
<td>f64vector-comparator</td>
<td>562</td>
</tr>
<tr>
<td>FD_DUPFD</td>
<td>433</td>
</tr>
<tr>
<td>F_GETFD</td>
<td>433</td>
</tr>
<tr>
<td>F_GETFL</td>
<td>433</td>
</tr>
<tr>
<td>F_GETLK</td>
<td>433</td>
</tr>
<tr>
<td>F_GETOWN</td>
<td>433</td>
</tr>
<tr>
<td>F_OK</td>
<td>303</td>
</tr>
<tr>
<td>F_RDLC</td>
<td>434</td>
</tr>
<tr>
<td>F_SETFD</td>
<td>433</td>
</tr>
<tr>
<td>F_SETFL</td>
<td>433</td>
</tr>
<tr>
<td>F_SETLK</td>
<td>433</td>
</tr>
<tr>
<td>F_SETLKW</td>
<td>433</td>
</tr>
<tr>
<td>F_SETOWN</td>
<td>433</td>
</tr>
<tr>
<td>F_UNLCK</td>
<td>434</td>
</tr>
<tr>
<td>F_ULOCK</td>
<td>434</td>
</tr>
<tr>
<td>false</td>
<td>786</td>
</tr>
<tr>
<td>family of &lt;sys-addinfo&gt;</td>
<td>477</td>
</tr>
<tr>
<td>fatal-handler of &lt;listener&gt;</td>
<td>456</td>
</tr>
<tr>
<td>favail of &lt;sys-statvfs&gt;</td>
<td>435</td>
</tr>
<tr>
<td>FD_CLOEXEC</td>
<td>433</td>
</tr>
<tr>
<td>ffree of &lt;sys-statvfs&gt;</td>
<td>435</td>
</tr>
<tr>
<td>file of &lt;info-node&gt;</td>
<td>1004</td>
</tr>
<tr>
<td>file-mode of &lt;dhm&gt;</td>
<td>872</td>
</tr>
<tr>
<td>FILE_SHARE_READ</td>
<td>899</td>
</tr>
<tr>
<td>FILE_SHARE_WRITE</td>
<td>899</td>
</tr>
<tr>
<td>filenumber of &lt;buffered-input-port&gt;</td>
<td>575</td>
</tr>
<tr>
<td>filenumber of &lt;buffered-output-port&gt;</td>
<td>576</td>
</tr>
<tr>
<td>files of &lt;sys-statvfs&gt;</td>
<td>435</td>
</tr>
<tr>
<td>fill of &lt;buffered-input-port&gt;</td>
<td>575</td>
</tr>
<tr>
<td>finalizer of &lt;listener&gt;</td>
<td>456</td>
</tr>
<tr>
<td>f1</td>
<td>683</td>
</tr>
<tr>
<td>f1-1/e</td>
<td>671</td>
</tr>
<tr>
<td>f1-1/log-10</td>
<td>671</td>
</tr>
<tr>
<td>f1-1/log-2</td>
<td>671</td>
</tr>
<tr>
<td>f1-1/log-phi</td>
<td>672</td>
</tr>
<tr>
<td>f1-1/pi</td>
<td>671</td>
</tr>
<tr>
<td>f1-1/sqrt-2</td>
<td>672</td>
</tr>
</tbody>
</table>
Appendix F: Variable Index

G

gauce-version of
<gauche-package-description> ............................ 479
GDBM_CACHESIZE .......................... 878
GDBM_CENTFREE ........................... 878
GDBM_COALESCEBLKS .......................... 878
GDBM_FAST .............................. 877
GDBM_FASTMODE ............................ 878
GDBM_INSERT .............................. 877
GDBM_NEWDB .............................. 877
GDBM_NOLock .............................. 877
GDBM_READER .............................. 877
GDBM_REPLACE .............................. 877
GDBM_SYNC .............................. 877
GDBM_SYNCHD .............................. 878
GDBM_WCREAT .............................. 877
GDBM_WRITER .............................. 877
gecos of <sys-passwd> ......................... 305
GENERIC_READ ............................. 899
GENERIC_WRITE ............................. 899
getb of <virtual-input-port> ................. 572
getc of <virtual-input-port> ................. 573
gets of <virtual-input-port> .................. 573
gid of <sys-group> .......................... 304
gid of <sys-passwd> .......................... 305
gid of <sys-stat> ........................... 302
G group/unchanged ........................... 746

H

h-file of <gen-unit> .......................... 383
hashmap-comparator .......................... 663
headers of <mine-part> ......................... 943
hmac-block-size of
<message-digest-algorithm-meta> ............ 1018
homepage of <gauche-package-description> .... 479
hour of <date> .............................. 703
hour of <sys-tm> .............................. 317

I

iflag of <sys-termios> ......................... 519
indent of <write-controls> ...................... 276
index of <mine-part> .......................... 943
init-epilogue of <gen-init> ....................... 383
init-prologue of <gen-unit> ....................... 383
initargs of <class> ........................... 353
ino of <sys-stat> ............................ 302
input-delay of <vt100> ........................ 986
input-port of <listener> ......................... 455
integer-comparator ............................ 121
iport of <vt100> .............................. 986
isdst of <sys-tm> ............................. 317

K

key-convert of <dbm> ............................ 872
key.ascii-char of <win-input-record> .......... 900
key.control-key-state of
<win-input-record> ............................ 900
key.down of <win-input-record> ............... 900
key.repeat-count of <win-input-record> ...... 900
key.unicode-char of <win-input-record> ...... 900
key.virtual-key-code of <win-input-record> .. 900
key.virtual-scan-code of
<win-input-record> ............................ 900

L

LC_ALL ........................................ 306
LC_COLLATE .................................... 306
LCCTYPE ....................................... 306
LC_MONETARY .................................. 306
LC_NUMERIC .................................... 306
LC_TIME ........................................ 306
len of <sys-flock> ............................. 434
length of <queue> .............................. 837
length of <write-controls> ....................... 276
level of <write-controls> ....................... 276
1flag of <sys-termios> ........................ 519
licenses of <gauche-package-description> .... 479
line of <read-error> ............................ 255
list-comparator ............................... 121
lock-file-name of <lock-file-failure> ......... 892
lock-policy of <log-drain> ..................... 459
log-drain of <ftp-connection> ................. 925

M

maintainers of
<gauche-package-description> .................. 479
mapping-comparator .......................... 659
max-length of <mtqueue> ....................... 837
maximum-window-size.x of
<win-console-screen-buffer-info> .......... 900
maximum-window-size.y of
<win-console-screen-buffer-info> .......... 900
mday of <sys-tm> ............................. 317
mem of <sys-group> ........................... 304
menu.command-id of <win-input-record> ....... 901
message of <message-condition> ............... 254
message of <parse-error> ...................... 908
min of <sys-tm> ............................. 316
minute of <date> .............................. 703
mode of <sys-stat> ............................ 302
mon of <sys-tm> .............................. 317
month of <date> ............................... 703
mouse.button-state of <win-input-record> ... 901
mouse.control-key-state of
<win-input-record> ............................ 901
mouse.event-flags of <win-input-record> .... 901
mouse.x of <win-input-record> ............... 901
mouse.y of <win-input-record> ............... 901
MSG_CTRUNC ................................. 473
MSG_DONTROUTE ............................... 473
MSG_ERROR .................................... 473
MSG_OOB ...................................... 473
MSG_PEEK ..................................... 473
MSG_TRUNC ..................................... 473
Appendix F: Variable Index 1131

MSG_WAITALL ........................................... 473
mtime of <sys-stat> ..................................... 302
mtime of <sys-stat> ..................................... 302
mutex of <abandoned-mutex-exception> ............. 544

N
name of <gen-unit> ........................................ 383
name of <class> .......................................... 352
name of <condition-variable> ....................... 539
name of <gauge-package-description> ............. 478
name of <info-node> .................................... 1004
name of <mutex> .......................................... 536
name of <sys-group> ..................................... 304
name of <sys-hostent> ................................... 475
name of <sys-passwd> ................................... 305
name of <sys-servent> ................................... 476
name of <thread> ......................................... 533
namemax of <sys-statvfs> ............................... 435
nanosecond of <date> ..................................... 703
nanosecond of <time> ..................................... 318
next of <info-node> ...................................... 1004
nil .............................................................. 786
nil .............................................................. 683
nlink of <sys-stat> ....................................... 302
nolock of <gdbm> ........................................... 876
nothing ....................................................... 683
num-instance-slots of <class> ....................... 353
number-comparator ....................................... 121

O
O_ACCMODE .................................................. 433
O_APPEND ................................................... 433
O_CLOEXEC .................................................. 433
O_CREAT ....................................................... 433
O_EXCL ......................................................... 433
O_NOCTTY ...................................................... 433
O_NOFOLLOW .................................................. 433
O_NONBLOCK ............................................... 433
O_RDONLY ................................................. 433
O_RDWR ........................................................ 433
O_TRUNC ........................................................ 433
O_WRONLY ...................................................... 433
object of <json-construct-error> .................... 938
objects of <parse-error> ................................. 908
oflag of <sys-termsios> ................................... 519
open/append ............................................... 745
open/create ................................................. 745
open/exclusive ............................................ 745
open/nofollow ............................................. 745
open/truncate ............................................. 745
oprt of <vt100> ............................................. 986
output ......................................................... 687
output-port of <listener> .............................. 456
owner/unchanged ......................................... 746

P
pad-char ...................................................... 687
pair-comparator .......................................... 121
pair-interval-protocol ................................ 1045
parameters of <time-part> ............................. 943
parent of <time-part> .................................... 943
passive of <ftp-connection> ......................... 925
passwd of <sys-group> .................................... 304
passwd of <sys-passwd> .................................. 305
path of <dbm> ............................................... 872
path of <log-drain> ....................................... 458
payload of <edn-object> ................................. 995
perm of <sys-stat> ........................................ 302
PF_INET ....................................................... 470
PF_INET6 ..................................................... 470
PF_UNIX ....................................................... 470
pi ............................................................... 893
pi/160 .......................................................... 893
pi/2 ............................................................ 893
pi/4 ............................................................ 893
pid of <sys-flock> ......................................... 434
pool of <thread-pool-shut-down> .................... 826
port ........................................................... 687
port of <port-error> ........................................ 256
port of <read-error> ....................................... 255
port of <sys-servent> .................................... 476
position of <json-parse-error> ...................... 937
position of <parse-error> .............................. 908
position of <read-error> ............................... 255
preamble of <cgen-unit> .................................. 383
precision ....................................................... 687
prefix of <log-drain> ..................................... 459
prepared of <dbi-query> ................................. 868
pretty of <write-controls> ............................. 276
prev of <info-node> ........................................ 1004
printer of <listener> ...................................... 456
process of <process-abnormal-exit> ................. 495
program-name of <log-drain> ......................... 459
prompter of <listener> ................................... 456
protection of <memory-region> ....................... 323
proto of <sys-servent> .................................. 476
protocol of <sys-addrinfo> ............................ 477
providing-modules of <gauge-package-description> 479
putc of <virtual-output-port> ......................... 574
putc of <virtual-output-port> ......................... 574
puts of <virtual-output-port> ......................... 574

R
R_OK .......................................................... 303
radius ........................................................ 687
radius of <write-controls> ............................. 276
RAND_MAX ..................................................... 325
rational-comparator .................................... 121
rdev of <sys-stat> ......................................... 302
reader of <listener> ...................................... 456
ready of <buffered-input-port> ....................... 575
ready of <virtual-input-port> ......................... 573
real of <time-result> ..................................... 547
real-comparator .......................................... 121
reason of <uncaught-exception> ..................... 544
redefined of <class> ...................................... 353
repository of <gauge-package-description> ......... 479
require of <gaucho-package-description> ..... 479
rest of <parse-error> ............................ 909
row .......................... 687
rw-mode of <dbm> ................................ 872

S
s16vector-comparator .......................... 562
s32vector-comparator .......................... 562
s64vector-comparator .......................... 562
svector-comparator .......................... 562
sec of <sys-tm> .................................. 316
second of <date> .................................. 703
second of <time> .................................. 318
seek of <buffered-input-port> .................. 575
seek of <buffered-output-port> ................. 576
seek of <virtual-input-port> ................. 573
seek of <virtual-output-port> ................. 574
set-comparator .................................. 614
shell of <sys-passwd> .............................. 305
SIGABRT ............................................. 307
SIGALRM ............................................. 307
SIGBUS ............................................. 307
SIGCHLD .............................................. 307
SIGCONT ............................................. 307
SIGFPE ............................................. 307
SIGIOT ............................................. 307
SIGKILL ............................................. 307
signal of <unhandled-signal-error> .......... 255
SIGPIPE .............................................. 307
SIGPOLL .............................................. 308
SIGPROF .............................................. 308
SIGQUIT ............................................. 308
SIGSEGV ............................................. 307
SIGSTKFLT ........................................... 307
SIGSTOP .............................................. 307
SIGTERM .............................................. 307
SIGTRAP .............................................. 307
SIGTSTP .............................................. 307
SIGTTIN .............................................. 307
SIGTTOU .............................................. 307
SIGURG ............................................. 307
SIGUSR1 ............................................. 307
SIGUSR2 ............................................. 307
SIGVTALRM .......................................... 308
SIGWINCH ............................................ 308
SIGXCPU ............................................. 307
SIGXFSZ ............................................. 307
size of <memory-region> .......................... 323
size of <sys-stat> .................................. 302
size.x of <win:console-screen-buffer-info> .... 900
size.y of <win:console-screen-buffer-info> .... 900
slots of <class> .................................... 852
SO_BROADCAST ....................................... 474
SO_ERROR ............................................ 475
SO_KEEPALIVE ....................................... 474
SO_OOBINLINE ....................................... 474
SO_PRIORITY ........................................ 475
SO_REUSEADDR ...................................... 474
SO_TYPE ............................................ 470
SOCK_DGRAM ........................................ 470
SOCK_RAW ........................................... 470
SOCK_STREAM ....................................... 470
socktype of <sys-addrinfo> ...................... 477
SOL_IP ............................................. 474
SOL_SOCKET ......................................... 474
SOL_TCP ............................................. 474
source of <mine-part> .............................. 943
span of <read-error> ............................ 255
specific of <condition-variable> ......... 539
specific of <mutex> ............................ 537
specific of <thread> ............................ 533
sql-string of <sql-parse-error> .......... 1012
ssax:Prefix-XML .................................. 963
SSL_OBJ_PKCS12 ................................... 949
SSL_OBJ_PKCS8 ................................... 949
SSL_OBJ_RSA_KEY ................................... 949
SSL_OBJ_X509_CACERT ............................. 949
SSL_OBJ_X509_CERT ................................ 949
ST_NOSUID .......................................... 435
ST_RDONLY .......................................... 435
start of <sys-flock> ............................ 434
state of <mutex> .................................. 536
STD_ERROR_HANDLE ............................... 902
STD_INPUT_HANDLE ............................... 902
STD_OUTPUT_HANDLE ............................... 902
stream-null ........................................ 1033
string-ci-comparator ............................ 120
string-comparator ............................... 120
string-length of <write-controls> .......... 277
string-width ....................................... 687
subtype of <mine-part> ........................... 943
sync of <gdbm> ..................................... 876
sys of <time-result> ............................. 547
syslog-facility of <log-drain> ................. 460
syslog-priority of <log-drain> ................. 460

T
tag of <edn-object> .................................. 995
TCIFLUSH ............................................ 519
TCIOFF ............................................ 520
TCIOLUSH ............................................ 519
TCION ............................................. 520
TCG0FF ............................................. 520
TCG0FF ............................................. 520
TCSADRAIN .......................................... 519
TCSAFLUSH .......................................... 519
TCSANOW ........................................... 519
terminator of <terminated-thread-exception> ... 544
textual-input ...................................... 745
textual-output .................................... 745
the-empty-stream .................................. 787
thread of <thread-exception> ................. 543
time-comparator ................................. 121
time-duration ...................................... 702
time-monotonic .................................... 702
time-process ...................................... 702
time-tai ........................................... 701
time-thread ....................................... 702
time-utc ........................................ 701
time/now ......................................... 746
time/unchanged .................................... 746
token of <parse-error> .......................... 909
transfer-encoding of <mime-part> ............... 943
transfer-type of <ftp-connection> ............... 925
true .............................................. 786
type of <mime-part> .............................. 943
type of <parse-error> .......................... 908
type of <sys-flock> .............................. 434
type of <sys-stat> .............................. 301
type of <time> .................................. 318

U
ul6vector-comparator .............................. 562
u32vector-comparator ............................. 562
u64vector-comparator ............................. 562
u8vector-comparator .............................. 562
uid of <sys-passwd> .............................. 305
uid of <sys-stat> ................................. 302
ulid-comparator .................................. 864
up of <info-node> .................................. 1004
user of <time-result> ............................ 547
uuid-comparator ................................. 954
uvector-comparator ................................ 121

V
value-convert of <dbm> ............................ 872
vector-comparator .................................. 121
version of <gauche-package-description> ....... 478

W
W_OK ............................................... 303
wday of <sys-tm> .................................. 317
whence of <sys-flock> ............................ 434
width ............................................. 687
width of <write-controls> ......................... 276
window-buffer-size.x of <win:input-record> .. 901
window-buffer-size.y of <win:input-record> .. 901
window.bottom of <win:console-screen-buffer-info> 900
window.left of <win:console-screen-buffer-info> 900
window.right of <win:console-screen-buffer-info> 900
window.top of <win:console-screen-buffer-info> 900
word-separator? .................................. 688
writer ............................................ 687

X
X_OK .............................................. 303

Y
yday of <sys-tm> .................................. 317
year of <date> .................................... 703
year of <sys-tm> .................................. 317

Z
Z_ASCII .......................................... 958
Z_BEST_COMPRESSION ............................. 956
Z_BEST_SPEED .................................... 956
Z_BINARY ......................................... 958
Z_DEFAULT_COMPRESSION .......................... 956
Z_DEFAULT_STRATEGY .............................. 957
Z_FILTERED ...................................... 957
Z_FIXED .......................................... 957
Z_HUFFMAN_ONLY .................................. 957
Z_NO_COMPRESSION ................................ 956
Z_RLE ............................................ 957
Z_TEXT .......................................... 958
Z_UNKNOWN ....................................... 958
zone-offset of <date> ............................. 703