This manual documents Guile version 2.2.6.


Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License.”
Table of Contents

Preface ......................................................... 1
  Contributors to this Manual ................................. 1
  The Guile License ............................................. 1

1 Introduction ..................................................... 3
  1.1 Guile and Scheme .............................................. 3
  1.2 Combining with C Code ....................................... 3
  1.3 Guile and the GNU Project ................................... 4
  1.4 Interactive Programming .................................... 4
  1.5 Supporting Multiple Languages ............................... 5
  1.6 Obtaining and Installing Guile ............................... 5
  1.7 Organisation of this Manual .................................. 6
  1.8 Typographical Conventions ................................... 6

2 Hello Guile! ...................................................... 9
  2.1 Running Guile Interactively .................................. 9
  2.2 Running Guile Scripts ......................................... 9
  2.3 Linking Guile into Programs .................................. 9
  2.4 Writing Guile Extensions ..................................... 10
  2.5 Using the Guile Module System ............................... 11
    2.5.1 Using Modules ............................................ 11
    2.5.2 Writing new Modules ..................................... 12
    2.5.3 Putting Extensions into Modules ......................... 12
  2.6 Reporting Bugs ............................................... 12

3 Hello Scheme! ................................................... 15
  3.1 Data Types, Values and Variables ............................ 15
    3.1.1 Latent Typing ............................................. 15
    3.1.2 Values and Variables ..................................... 15
    3.1.3 Defining and Setting Variables ......................... 16
  3.2 The Representation and Use of Procedures .................... 17
    3.2.1 Procedures as Values ..................................... 17
    3.2.2 Simple Procedure Invocation ............................. 18
    3.2.3 Creating and Using a New Procedure ..................... 19
    3.2.4 Lambda Alternatives ..................................... 20
  3.3 Expressions and Evaluation ................................... 20
    3.3.1 Evaluating Expressions and Executing Programs .......... 21
      3.3.1.1 Evaluating Literal Data ............................... 21
      3.3.1.2 Evaluating a Variable Reference ..................... 22
      3.3.1.3 Evaluating a Procedure Invocation Expression .......... 22
      3.3.1.4 Evaluating Special Syntactic Expressions ............. 23
    3.3.2 Tail calls ............................................... 24
5 Programming in C ........................................ 59

5.1 Parallel Installations .................................... 59
5.2 Linking Programs With Guile ............................. 60
  5.2.1 Guile Initialization Functions ....................... 60
  5.2.2 A Sample Guile Main Program ....................... 61
  5.2.3 Building the Example with Make .................... 61
  5.2.4 Building the Example with Autoconf ................. 62
5.3 Linking Guile with Libraries ............................ 63
  5.3.1 A Sample Guile Extension ........................... 64
5.4 General concepts for using libguile ..................... 64
  5.4.1 Dynamic Types ..................................... 65
  5.4.2 Garbage Collection .................................. 67
  5.4.3 Control Flow ....................................... 68
  5.4.4 Asynchronous Signals ................................ 70
  5.4.5 Multi-Threading .................................... 71
5.5 Defining New Foreign Object Types ..................... 74
  5.5.1 Defining Foreign Object Types ...................... 74
  5.5.2 Creating Foreign Objects ......................... 75
  5.5.3 Type Checking of Foreign Objects .................. 76
  5.5.4 Foreign Object Memory Management ................. 77
  5.5.5 Foreign Objects and Scheme ......................... 80
5.6 Function Snarfing ...................................... 81
5.7 An Overview of Guile Programming ....................... 83
  5.7.1 How One Might Extend Dia Using Guile ............... 83
    5.7.1.1 Deciding Why You Want to Add Guile .......... 84
    5.7.1.2 Four Steps Required to Add Guile ............ 84
    5.7.1.3 How to Represent Dia Data in Scheme ........ 85
    5.7.1.4 Writing Guile Primitives for Dia ............. 86
    5.7.1.5 Providing a Hook for the Evaluation of Scheme Code ........ 87
    5.7.1.6 Top-level Structure of Guile-enabled Dia ...... 88
    5.7.1.7 Going Further with Dia and Guile ............ 88
  5.7.2 Why Scheme is More Hackable Than C ................. 90
  5.7.3 Example: Using Guile for an Application Testbed .... 90
  5.7.4 A Choice of Programming Options .................... 91
    5.7.4.1 What Functionality is Already Available? ...... 92
    5.7.4.2 Functional and Performance Constraints .......... 92
    5.7.4.3 Your Preferred Programming Style ............. 92
    5.7.4.4 What Controls Program Execution? ............. 92
  5.7.5 How About Application Users? ....................... 92
5.8 Autoconf Support ........................................ 94
  5.8.1 Autoconf Background ................................ 94
  5.8.2 Autoconf Macros .................................... 94
  5.8.3 Using Autoconf Macros .............................. 96
## 6 API Reference

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Overview of the Guile API</td>
<td>99</td>
</tr>
<tr>
<td>6.2 Deprecation</td>
<td>100</td>
</tr>
<tr>
<td>6.3 The SCM Type</td>
<td>100</td>
</tr>
<tr>
<td>6.4 Initializing Guile</td>
<td>101</td>
</tr>
<tr>
<td>6.5 Snarfing Macros</td>
<td>102</td>
</tr>
<tr>
<td>6.6 Data Types</td>
<td>104</td>
</tr>
<tr>
<td>6.6.1 Booleans</td>
<td>104</td>
</tr>
<tr>
<td>6.6.2 Numerical data types</td>
<td>105</td>
</tr>
<tr>
<td>6.6.2.1 Scheme’s Numerical “Tower”</td>
<td>105</td>
</tr>
<tr>
<td>6.6.2.2 Integers</td>
<td>106</td>
</tr>
<tr>
<td>6.6.2.3 Real and Rational Numbers</td>
<td>110</td>
</tr>
<tr>
<td>6.6.2.4 Complex Numbers</td>
<td>113</td>
</tr>
<tr>
<td>6.6.2.5 Exact and Inexact Numbers</td>
<td>113</td>
</tr>
<tr>
<td>6.6.2.6 Read Syntax for Numerical Data</td>
<td>115</td>
</tr>
<tr>
<td>6.6.2.7 Operations on Integer Values</td>
<td>116</td>
</tr>
<tr>
<td>6.6.2.8 Comparison Predicates</td>
<td>117</td>
</tr>
<tr>
<td>6.6.2.9 Converting Numbers To and From Strings</td>
<td>118</td>
</tr>
<tr>
<td>6.6.2.10 Complex Number Operations</td>
<td>118</td>
</tr>
<tr>
<td>6.6.2.11 Arithmetic Functions</td>
<td>119</td>
</tr>
<tr>
<td>6.6.2.12 Scientific Functions</td>
<td>123</td>
</tr>
<tr>
<td>6.6.2.13 Bitwise Operations</td>
<td>125</td>
</tr>
<tr>
<td>6.6.2.14 Random Number Generation</td>
<td>127</td>
</tr>
<tr>
<td>6.6.3 Characters</td>
<td>129</td>
</tr>
<tr>
<td>6.6.4 Character Sets</td>
<td>134</td>
</tr>
<tr>
<td>6.6.4.1 Character Set Predicates/Comparison</td>
<td>134</td>
</tr>
<tr>
<td>6.6.4.2 Iterating Over Character Sets</td>
<td>134</td>
</tr>
<tr>
<td>6.6.4.3 Creating Character Sets</td>
<td>135</td>
</tr>
<tr>
<td>6.6.4.4 Querying Character Sets</td>
<td>137</td>
</tr>
<tr>
<td>6.6.4.5 Character-Set Algebra</td>
<td>138</td>
</tr>
<tr>
<td>6.6.4.6 Standard Character Sets</td>
<td>139</td>
</tr>
<tr>
<td>6.6.5 Strings</td>
<td>141</td>
</tr>
<tr>
<td>6.6.5.1 String Read Syntax</td>
<td>141</td>
</tr>
<tr>
<td>6.6.5.2 String Predicates</td>
<td>143</td>
</tr>
<tr>
<td>6.6.5.3 String Constructors</td>
<td>144</td>
</tr>
<tr>
<td>6.6.5.4 List/String conversion</td>
<td>145</td>
</tr>
<tr>
<td>6.6.5.5 String Selection</td>
<td>145</td>
</tr>
<tr>
<td>6.6.5.6 String Modification</td>
<td>147</td>
</tr>
<tr>
<td>6.6.5.7 String Comparison</td>
<td>148</td>
</tr>
<tr>
<td>6.6.5.8 String Searching</td>
<td>152</td>
</tr>
<tr>
<td>6.6.5.9 Alphabetic Case Mapping</td>
<td>154</td>
</tr>
<tr>
<td>6.6.5.10 Reversing and Appending Strings</td>
<td>155</td>
</tr>
<tr>
<td>6.6.5.11 Mapping, Folding, and Unfolding</td>
<td>156</td>
</tr>
<tr>
<td>6.6.5.12 Miscellaneous String Operations</td>
<td>158</td>
</tr>
<tr>
<td>6.6.5.13 Representing Strings as Bytes</td>
<td>159</td>
</tr>
<tr>
<td>6.6.5.14 Conversion to/from C</td>
<td>160</td>
</tr>
<tr>
<td>6.6.5.15 String Internals</td>
<td>163</td>
</tr>
<tr>
<td>6.6.6 Symbols</td>
<td>164</td>
</tr>
</tbody>
</table>
6.6.6.1 Symbols as Discrete Data ........................................... 165
6.6.6.2 Symbols as Lookup Keys ........................................... 166
6.6.6.3 Symbols as Denoting Variables ................................... 167
6.6.6.4 Operations Related to Symbols ................................... 167
6.6.6.5 Function Slots and Property Lists ............................... 170
6.6.6.6 Extended Read Syntax for Symbols ............................... 171
6.6.6.7 Uninterned Symbols .................................................. 172
6.6.7 Keywords ...................................................................... 174
6.6.7.1 Why Use Keywords? ................................................... 174
6.6.7.2 Coding With Keywords ............................................... 174
6.6.7.3 Keyword Read Syntax ............................................... 175
6.6.7.4 Keyword Procedures .................................................. 176
6.6.8 Pairs ............................................................................ 178
6.6.9 Lists ............................................................................ 181
6.6.9.1 List Read Syntax ....................................................... 181
6.6.9.2 List Predicates .......................................................... 181
6.6.9.3 List Constructors ....................................................... 182
6.6.9.4 List Selection ............................................................ 183
6.6.9.5 Append and Reverse ................................................... 183
6.6.9.6 List Modification ....................................................... 184
6.6.9.7 List Searching ........................................................... 185
6.6.9.8 List Mapping ............................................................. 186
6.6.10 Vectors ...................................................................... 186
6.6.10.1 Read Syntax for Vectors ............................................ 187
6.6.10.2 Dynamic Vector Creation and Validation ..................... 187
6.6.10.3 Accessing and Modifying Vector Contents ..................... 188
6.6.10.4 Vector Accessing from C .......................................... 189
6.6.10.5 Uniform Numeric Vectors .......................................... 190
6.6.11 Bit Vectors ................................................................. 191
6.6.12 Bytevectors ................................................................. 193
6.6.12.1 Endianness ............................................................... 193
6.6.12.2 Manipulating Bytevectors ......................................... 194
6.6.12.3 Interpreting Bytevector Contents as Integers ............... 195
6.6.12.4 Converting Bytevectors to/from Integer Lists ............. 197
6.6.12.5 Interpreting Bytevector Contents as Floating Point Numbers .................................................. 198
6.6.12.6 Interpreting Bytevector Contents as Unicode Strings ............... 199
6.6.12.7 Accessing Bytevectors with the Array API .................. 200
6.6.12.8 Accessing Bytevectors with the SRFI-4 API ................. 200
6.6.13 Arrays ...................................................................... 200
6.6.13.1 Array Syntax ........................................................... 201
6.6.13.2 Array Procedures ..................................................... 202
6.6.13.3 Shared Arrays ......................................................... 206
6.6.13.4 Arrays as arrays of arrays ....................................... 208
6.6.13.5 Accessing Arrays from C ......................................... 210
6.6.14 VLists ..................................................................... 215
6.6.15 Record Overview ........................................................ 217
6.6.16 SRFI-9 Records .......................................................... 218
Non-toplevel Record Definitions ........................................ 219
Custom Printers ......................................................... 219
Functional “Setters” .................................................... 219
6.6.17 Records ................................................................ 221
6.6.18 Structures ............................................................ 222
  6.6.18.1 Vtables ............................................................ 223
  6.6.18.2 Structure Basics ............................................... 223
  6.6.18.3 Vtable Contents ............................................... 225
  6.6.18.4 Meta-Vtables .................................................. 226
  6.6.18.5 Vtable Example ............................................... 227
6.6.19 Dictionary Types ..................................................... 229
6.6.20 Association Lists ..................................................... 229
  6.6.20.1 Alist Key Equality ............................................ 230
  6.6.20.2 Adding or Setting Alist Entries .............................. 230
  6.6.20.3 Retrieving Alist Entries ..................................... 232
  6.6.20.4 Removing Alist Entries ...................................... 233
  6.6.20.5 Sloppy Alist Functions ...................................... 234
  6.6.20.6 Alist Example ................................................ 235
6.6.21 VList-Based Hash Lists or “VHashes” ......................... 235
6.6.22 Hash Tables ........................................................... 237
  6.6.22.1 Hash Table Examples ......................................... 238
  6.6.22.2 Hash Table Reference ....................................... 239
6.6.23 Other Types ............................................................. 243
  6.7 Foreign Objects ......................................................... 243
6.8 Smobs ......................................................................... 245
6.9 Procedures ............................................................... 248
  6.9.1 Lambda: Basic Procedure Creation ............................ 248
  6.9.2 Primitive Procedures ............................................ 249
  6.9.3 Compiled Procedures ............................................ 250
  6.9.4 Optional Arguments ............................................. 252
    6.9.4.1 lambda* and define* ....................................... 252
    6.9.4.2 (ice-9 optargs) ............................................ 254
  6.9.5 Case-lambda ......................................................... 255
  6.9.6 Higher-Order Functions ......................................... 257
  6.9.7 Procedure Properties and Meta-information ................. 258
  6.9.8 Procedures with Setters ......................................... 259
  6.9.9 Inlinable Procedures ............................................. 260
6.10 Macros ...................................................................... 261
  6.10.1 Defining Macros .................................................. 261
  6.10.2 Syntax-rules Macros ............................................. 262
    6.10.2.1 Patterns .................................................... 263
    6.10.2.2 Hygiene ..................................................... 265
    6.10.2.3 Shorthands .................................................. 266
    6.10.2.4 Reporting Syntax Errors in Macros .................... 266
    6.10.2.5 Specifying a Custom Ellipsis Identifier ................ 266
    6.10.2.6 Further Information ....................................... 267
  6.10.3 Support for the syntax-case System ......................... 267
    6.10.3.1 Why syntax-case? ......................................... 269
### 6.10.3.2 Custom Ellipsis Identifiers for syntax-case Macros

272

### 6.10.4 Syntax Transformer Helpers

272

### 6.10.5 Lisp-style Macro Definitions

275

### 6.10.6 Identifier Macros

276

### 6.10.7 Syntax Parameters

278

### 6.10.8 Eval-when

279

### 6.10.9 Macro Expansion

280

### 6.10.10 Hygiene and the Top-Level

281

### 6.10.11 Internal Macros

282

### 6.11 General Utility Functions

283

### 6.11.1 Equality

283

### 6.11.2 Object Properties

285

### 6.11.3 Sorting

286

### 6.11.4 Copying Deep Structures

288

### 6.11.5 General String Conversion

288

### 6.11.6 Hooks

288

#### 6.11.6.1 Hook Usage by Example

289

#### 6.11.6.2 Hook Reference

289

#### 6.11.6.3 Hooks For C Code

291

#### 6.11.6.4 Hooks for Garbage Collection

292

#### 6.11.6.5 Hooks into the Guile REPL

293

### 6.12 Definitions and Variable Bindings

293

### 6.12.1 Top Level Variable Definitions

293

### 6.12.2 Local Variable Bindings

295

### 6.12.3 Internal definitions

296

### 6.12.4 Querying variable bindings

297

### 6.12.5 Binding multiple return values

297

### 6.13 Controlling the Flow of Program Execution

298

#### 6.13.1 Sequencing and Splicing

298

#### 6.13.2 Simple Conditional Evaluation

299

#### 6.13.3 Conditional Evaluation of a Sequence of Expressions

301

#### 6.13.4 Iteration mechanisms

301

#### 6.13.5 Prompts

303

##### 6.13.5.1 Prompt Primitives

303

##### 6.13.5.2 Shift, Reset, and All That

306

#### 6.13.6 Continuations

307

#### 6.13.7 Returning and Accepting Multiple Values

309

#### 6.13.8 Exceptions

311

##### 6.13.8.1 Exception Terminology

311

##### 6.13.8.2 Catching Exceptions

311

##### 6.13.8.3 Throw Handlers

314

##### 6.13.8.4 Throwing Exceptions

315

##### 6.13.8.5 How Guile Implements Exceptions

316

#### 6.13.9 Procedures for Signaling Errors

316

#### 6.13.10 Dynamic Wind

317

#### 6.13.11 Fluids and Dynamic States

321

#### 6.13.12 Parameters

324

#### 6.13.13 How to Handle Errors

325
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.18.7 Load Paths</td>
<td>389</td>
</tr>
<tr>
<td>6.18.8 Character Encoding of Source Files</td>
<td>392</td>
</tr>
<tr>
<td>6.18.9 Delayed Evaluation</td>
<td>393</td>
</tr>
<tr>
<td>6.18.10 Local Evaluation</td>
<td>393</td>
</tr>
<tr>
<td>6.18.11 Local Inclusion</td>
<td>394</td>
</tr>
<tr>
<td>6.18.12 Sandboxed Evaluation</td>
<td>395</td>
</tr>
<tr>
<td>6.18.13 REPL Servers</td>
<td>399</td>
</tr>
<tr>
<td>6.18.14 Cooperative REPL Servers</td>
<td>400</td>
</tr>
<tr>
<td>6.19 Memory Management and Garbage Collection</td>
<td>400</td>
</tr>
<tr>
<td>6.19.1 Function related to Garbage Collection</td>
<td>401</td>
</tr>
<tr>
<td>6.19.2 Memory Blocks</td>
<td>402</td>
</tr>
<tr>
<td>6.19.3 Weak References</td>
<td>404</td>
</tr>
<tr>
<td>6.19.3.1 Weak hash tables</td>
<td>404</td>
</tr>
<tr>
<td>6.19.3.2 Weak vectors</td>
<td>405</td>
</tr>
<tr>
<td>6.19.4 Guardians</td>
<td>405</td>
</tr>
<tr>
<td>6.20 Modules</td>
<td>406</td>
</tr>
<tr>
<td>6.20.1 General Information about Modules</td>
<td>407</td>
</tr>
<tr>
<td>6.20.2 Using Guile Modules</td>
<td>407</td>
</tr>
<tr>
<td>6.20.3 Creating Guile Modules</td>
<td>409</td>
</tr>
<tr>
<td>6.20.4 Modules and the File System</td>
<td>412</td>
</tr>
<tr>
<td>6.20.5 R6RS Version References</td>
<td>413</td>
</tr>
<tr>
<td>6.20.6 R6RS Libraries</td>
<td>414</td>
</tr>
<tr>
<td>6.20.7 Variables</td>
<td>415</td>
</tr>
<tr>
<td>6.20.8 Module System Reflection</td>
<td>416</td>
</tr>
<tr>
<td>6.20.9 Accessing Modules from C</td>
<td>418</td>
</tr>
<tr>
<td>6.20.10 provide and require</td>
<td>421</td>
</tr>
<tr>
<td>6.20.11 Environments</td>
<td>421</td>
</tr>
<tr>
<td>6.21 Foreign Function Interface</td>
<td>422</td>
</tr>
<tr>
<td>6.21.1 Foreign Libraries</td>
<td>422</td>
</tr>
<tr>
<td>6.21.2 Foreign Functions</td>
<td>423</td>
</tr>
<tr>
<td>6.21.3 C Extensions</td>
<td>425</td>
</tr>
<tr>
<td>6.21.4 Modules and Extensions</td>
<td>426</td>
</tr>
<tr>
<td>6.21.5 Foreign Pointers</td>
<td>427</td>
</tr>
<tr>
<td>6.21.5.1 Foreign Types</td>
<td>427</td>
</tr>
<tr>
<td>6.21.5.2 Foreign Variables</td>
<td>428</td>
</tr>
<tr>
<td>6.21.5.3 Void Pointers and Byte Access</td>
<td>430</td>
</tr>
<tr>
<td>6.21.5.4 Foreign Structs</td>
<td>432</td>
</tr>
<tr>
<td>6.21.6 Dynamic FFI</td>
<td>433</td>
</tr>
<tr>
<td>6.22 Threads, Mutexes, Asynces and Dynamic Roots</td>
<td>436</td>
</tr>
<tr>
<td>6.22.1 Threads</td>
<td>436</td>
</tr>
<tr>
<td>6.22.2 Thread-Local Variables</td>
<td>438</td>
</tr>
<tr>
<td>6.22.3 Asynchronous Interrupts</td>
<td>439</td>
</tr>
<tr>
<td>6.22.4 Atomics</td>
<td>441</td>
</tr>
<tr>
<td>6.22.5 Mutexes and Condition Variables</td>
<td>442</td>
</tr>
<tr>
<td>6.22.6 Blocking in Guile Mode</td>
<td>446</td>
</tr>
<tr>
<td>6.22.7 Futures</td>
<td>447</td>
</tr>
<tr>
<td>6.22.8 Parallel forms</td>
<td>448</td>
</tr>
<tr>
<td>6.23 Configuration, Features and Runtime Options</td>
<td>450</td>
</tr>
</tbody>
</table>
6.23.1 Configuration, Build and Installation ...................... 450
6.23.2 Feature Tracking ............................................. 452
  6.23.2.1 Feature Manipulation .................................. 452
  6.23.2.2 Common Feature Symbols .............................. 452
6.23.3 Runtime Options ............................................ 454
  6.23.3.1 Examples of option use .................................. 454
6.24 Support for Other Languages ................................ 455
  6.24.1 Using Other Languages .................................... 455
  6.24.2 Emacs Lisp .................................................. 456
    6.24.2.1 Nil ..................................................... 456
    6.24.2.2 Dynamic Binding ..................................... 458
    6.24.2.3 Other Elisp Features ................................ 458
  6.24.3 ECMAScript ................................................ 458
6.25 Support for Internationalization ............................. 459
  6.25.1 Internationalization with Guile .......................... 459
  6.25.2 Text Collation ............................................ 460
  6.25.3 Character Case Mapping .................................. 461
  6.25.4 Number Input and Output .................................. 462
  6.25.5 Accessing Locale Information ............................ 462
  6.25.6 Gettext Support .......................................... 466
6.26 Debugging Infrastructure ..................................... 468
  6.26.1 Evaluation and the Scheme Stack .......................... 468
    6.26.1.1 Stack Capture ...................................... 468
    6.26.1.2 Stacks .............................................. 469
    6.26.1.3 Frames ............................................... 470
  6.26.2 Source Properties ....................................... 471
  6.26.3 Programmatic Error Handling ............................. 472
    6.26.3.1 Catching Exceptions ................................ 472
    6.26.3.2 Capturing the full error stack ..................... 474
    6.26.3.3 Pre-Unwind Debugging .............................. 476
    6.26.3.4 Stack Overflow ..................................... 477
    6.26.3.5 Debug options ...................................... 480
  6.26.4 Traps ...................................................... 480
    6.26.4.1 VM Hooks ............................................ 481
    6.26.4.2 Trap Interface ....................................... 482
    6.26.4.3 Low-Level Traps .................................... 483
    6.26.4.4 Tracing Traps ....................................... 485
    6.26.4.5 Trap States ......................................... 487
    6.26.4.6 High-Level Traps .................................... 487
  6.26.5 GDB Support ............................................. 489
6.27 Code Coverage Reports ....................................... 489

7 Guile Modules .................................................. 491
  7.1 SLIB ......................................................... 491
    7.1.1 SLIB installation ........................................ 491
    7.1.2 JACAL ..................................................... 492
  7.2 POSIX System Calls and Networking .......................... 492
    7.2.1 POSIX Interface Conventions ............................ 492
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.2</td>
<td>Ports and File Descriptors</td>
<td>493</td>
</tr>
<tr>
<td>7.2.3</td>
<td>File System</td>
<td>500</td>
</tr>
<tr>
<td>7.2.4</td>
<td>User Information</td>
<td>507</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Time</td>
<td>509</td>
</tr>
<tr>
<td>7.2.6</td>
<td>Runtime Environment</td>
<td>512</td>
</tr>
<tr>
<td>7.2.7</td>
<td>Processes</td>
<td>514</td>
</tr>
<tr>
<td>7.2.8</td>
<td>Signals</td>
<td>520</td>
</tr>
<tr>
<td>7.2.9</td>
<td>Terminals and Ptys</td>
<td>523</td>
</tr>
<tr>
<td>7.2.10</td>
<td>Pipes</td>
<td>524</td>
</tr>
<tr>
<td>7.2.11</td>
<td>Networking</td>
<td>525</td>
</tr>
<tr>
<td>7.2.11.1</td>
<td>Network Address Conversion</td>
<td>526</td>
</tr>
<tr>
<td>7.2.11.2</td>
<td>Network Databases</td>
<td>527</td>
</tr>
<tr>
<td>7.2.11.3</td>
<td>Network Socket Address</td>
<td>533</td>
</tr>
<tr>
<td>7.2.11.4</td>
<td>Network Sockets and Communication</td>
<td>535</td>
</tr>
<tr>
<td>7.2.11.5</td>
<td>Network Socket Examples</td>
<td>540</td>
</tr>
<tr>
<td>7.2.12</td>
<td>System Identification</td>
<td>541</td>
</tr>
<tr>
<td>7.2.13</td>
<td>Locales</td>
<td>542</td>
</tr>
<tr>
<td>7.2.14</td>
<td>Encryption</td>
<td>543</td>
</tr>
<tr>
<td>7.3</td>
<td>HTTP, the Web, and All That</td>
<td>543</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Types and the Web</td>
<td>543</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Universal Resource Identifiers</td>
<td>545</td>
</tr>
<tr>
<td>7.3.3</td>
<td>The Hyper-Text Transfer Protocol</td>
<td>548</td>
</tr>
<tr>
<td>7.3.4</td>
<td>HTTP Headers</td>
<td>551</td>
</tr>
<tr>
<td>7.3.4.1</td>
<td>HTTP Header Types</td>
<td>551</td>
</tr>
<tr>
<td>7.3.4.2</td>
<td>General Headers</td>
<td>552</td>
</tr>
<tr>
<td>7.3.4.3</td>
<td>Entity Headers</td>
<td>553</td>
</tr>
<tr>
<td>7.3.4.4</td>
<td>Request Headers</td>
<td>554</td>
</tr>
<tr>
<td>7.3.4.5</td>
<td>Response Headers</td>
<td>557</td>
</tr>
<tr>
<td>7.3.5</td>
<td>Transfer Codings</td>
<td>558</td>
</tr>
<tr>
<td>7.3.6</td>
<td>HTTP Requests</td>
<td>559</td>
</tr>
<tr>
<td>7.3.6.1</td>
<td>An Important Note on Character Sets</td>
<td>559</td>
</tr>
<tr>
<td>7.3.6.2</td>
<td>Request API</td>
<td>560</td>
</tr>
<tr>
<td>7.3.7</td>
<td>HTTP Responses</td>
<td>562</td>
</tr>
<tr>
<td>7.3.8</td>
<td>Web Client</td>
<td>564</td>
</tr>
<tr>
<td>7.3.9</td>
<td>Web Server</td>
<td>565</td>
</tr>
<tr>
<td>7.3.10</td>
<td>Web Examples</td>
<td>568</td>
</tr>
<tr>
<td>7.3.10.1</td>
<td>Hello, World!</td>
<td>568</td>
</tr>
<tr>
<td>7.3.10.2</td>
<td>Inspecting the Request</td>
<td>569</td>
</tr>
<tr>
<td>7.3.10.3</td>
<td>Higher-Level Interfaces</td>
<td>570</td>
</tr>
<tr>
<td>7.3.10.4</td>
<td>Conclusion</td>
<td>571</td>
</tr>
<tr>
<td>7.4</td>
<td>The (ice-9 getopt-long) Module</td>
<td>571</td>
</tr>
<tr>
<td>7.4.1</td>
<td>A Short getopt-long Example</td>
<td>572</td>
</tr>
<tr>
<td>7.4.2</td>
<td>How to Write an Option Specification</td>
<td>573</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Expected Command Line Format</td>
<td>574</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Reference Documentation for getopt-long</td>
<td>575</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Reference Documentation for option-ref</td>
<td>576</td>
</tr>
<tr>
<td>7.5</td>
<td>SRFI Support Modules</td>
<td>576</td>
</tr>
<tr>
<td>7.5.1</td>
<td>About SRFI Usage</td>
<td>576</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>7.5.2</td>
<td>SRFI-0 - cond-expand</td>
<td></td>
</tr>
<tr>
<td>7.5.3</td>
<td>SRFI-1 - List library</td>
<td></td>
</tr>
<tr>
<td>7.5.3.1</td>
<td>Constructors</td>
<td></td>
</tr>
<tr>
<td>7.5.3.2</td>
<td>Predicates</td>
<td></td>
</tr>
<tr>
<td>7.5.3.3</td>
<td>Selectors</td>
<td></td>
</tr>
<tr>
<td>7.5.3.4</td>
<td>Length, Append, Concatenate, etc.</td>
<td></td>
</tr>
<tr>
<td>7.5.3.5</td>
<td>Fold, Unfold &amp; Map</td>
<td></td>
</tr>
<tr>
<td>7.5.3.6</td>
<td>Filtering and Partitioning</td>
<td></td>
</tr>
<tr>
<td>7.5.3.7</td>
<td>Searching</td>
<td></td>
</tr>
<tr>
<td>7.5.3.8</td>
<td>Deleting</td>
<td></td>
</tr>
<tr>
<td>7.5.3.9</td>
<td>Association Lists</td>
<td></td>
</tr>
<tr>
<td>7.5.3.10</td>
<td>Set Operations on Lists</td>
<td></td>
</tr>
<tr>
<td>7.5.4</td>
<td>SRFI-2 - and-let*</td>
<td></td>
</tr>
<tr>
<td>7.5.5</td>
<td>SRFI-4 - Homogeneous numeric vector datatypes</td>
<td></td>
</tr>
<tr>
<td>7.5.5.1</td>
<td>SRFI-4 - Overview</td>
<td></td>
</tr>
<tr>
<td>7.5.5.2</td>
<td>SRFI-4 - API</td>
<td></td>
</tr>
<tr>
<td>7.5.5.3</td>
<td>SRFI-4 - Relation to bytevectors</td>
<td></td>
</tr>
<tr>
<td>7.5.5.4</td>
<td>SRFI-4 - Guile extensions</td>
<td></td>
</tr>
<tr>
<td>7.5.6</td>
<td>SRFI-6 - Basic String Ports</td>
<td></td>
</tr>
<tr>
<td>7.5.7</td>
<td>SRFI-8 - receive</td>
<td></td>
</tr>
<tr>
<td>7.5.8</td>
<td>SRFI-9 - define-record-type</td>
<td></td>
</tr>
<tr>
<td>7.5.9</td>
<td>SRFI-10 - Hash-Comma Reader Extension</td>
<td></td>
</tr>
<tr>
<td>7.5.10</td>
<td>SRFI-11 - let-values</td>
<td></td>
</tr>
<tr>
<td>7.5.11</td>
<td>SRFI-13 - String Library</td>
<td></td>
</tr>
<tr>
<td>7.5.12</td>
<td>SRFI-14 - Character-set Library</td>
<td></td>
</tr>
<tr>
<td>7.5.13</td>
<td>SRFI-16 - case-lambda</td>
<td></td>
</tr>
<tr>
<td>7.5.14</td>
<td>SRFI-17 - Generalized set!</td>
<td></td>
</tr>
<tr>
<td>7.5.15</td>
<td>SRFI-18 - Multithreading support</td>
<td></td>
</tr>
<tr>
<td>7.5.15.1</td>
<td>SRFI-18 Threads</td>
<td></td>
</tr>
<tr>
<td>7.5.15.2</td>
<td>SRFI-18 Mutexes</td>
<td></td>
</tr>
<tr>
<td>7.5.15.3</td>
<td>SRFI-18 Condition variables</td>
<td></td>
</tr>
<tr>
<td>7.5.15.4</td>
<td>SRFI-18 Time</td>
<td></td>
</tr>
<tr>
<td>7.5.15.5</td>
<td>SRFI-18 Exceptions</td>
<td></td>
</tr>
<tr>
<td>7.5.16</td>
<td>SRFI-19 - Time/Date Library</td>
<td></td>
</tr>
<tr>
<td>7.5.16.1</td>
<td>SRFI-19 Introduction</td>
<td></td>
</tr>
<tr>
<td>7.5.16.2</td>
<td>SRFI-19 Time</td>
<td></td>
</tr>
<tr>
<td>7.5.16.3</td>
<td>SRFI-19 Date</td>
<td></td>
</tr>
<tr>
<td>7.5.16.4</td>
<td>SRFI-19 Time/Date conversions</td>
<td></td>
</tr>
<tr>
<td>7.5.16.5</td>
<td>SRFI-19 Date to string</td>
<td></td>
</tr>
<tr>
<td>7.5.16.6</td>
<td>SRFI-19 String to date</td>
<td></td>
</tr>
<tr>
<td>7.5.17</td>
<td>SRFI-23 - Error Reporting</td>
<td></td>
</tr>
<tr>
<td>7.5.18</td>
<td>SRFI-26 - specializing parameters</td>
<td></td>
</tr>
<tr>
<td>7.5.19</td>
<td>SRFI-27 - Sources of Random Bits</td>
<td></td>
</tr>
<tr>
<td>7.5.19.1</td>
<td>The Default Random Source</td>
<td></td>
</tr>
<tr>
<td>7.5.19.2</td>
<td>Random Sources</td>
<td></td>
</tr>
<tr>
<td>7.5.19.3</td>
<td>Obtaining random number generator procedures</td>
<td></td>
</tr>
<tr>
<td>7.5.20</td>
<td>SRFI-28 - Basic Format Strings</td>
<td></td>
</tr>
<tr>
<td>7.5.21</td>
<td>SRFI-30 - Nested Multi-line Comments</td>
<td></td>
</tr>
</tbody>
</table>
7.6.2.6 rnrs sorting ........................................ 660
7.6.2.7 rnrs control ........................................ 660
7.6.2.8 R6RS Records ....................................... 661
7.6.2.9 rnrs records syntactic ............................... 662
7.6.2.10 rnrs records procedural ............................. 663
7.6.2.11 rnrs records inspection ............................. 664
7.6.2.12 rnrs exceptions ..................................... 665
7.6.2.13 rnrs conditions ...................................... 666
7.6.2.14 I/O Conditions ...................................... 668
7.6.2.15 Transcoders ......................................... 670
7.6.2.16 rnrs io ports ........................................ 672
7.6.2.17 R6RS File Ports ..................................... 676
7.6.2.18 rnrs io simple ....................................... 677
7.6.2.19 rnrs files ........................................... 679
7.6.2.20 rnrs programs ........................................ 679
7.6.2.21 rnrs arithmetic fixnums ............................. 679
7.6.2.22 rnrs arithmetic flonums ............................ 682
7.6.2.23 rnrs arithmetic bitwise ............................. 684
7.6.2.24 rnrs syntax-case ..................................... 685
7.6.2.25 rnrs hashtables ...................................... 686
7.6.2.26 rnrs enum ............................................ 688
7.6.2.27 rnrs .................................................. 690
7.6.2.28 rnrs eval ............................................ 690
7.6.2.29 rnrs mutable-pairs .................................. 690
7.6.2.30 rnrs mutable-strings ................................. 690
7.6.2.31 rnrs r5rs ............................................ 690

7.7 Pattern Matching ........................................ 691

7.8 Readline Support ......................................... 695
7.8.1 Loading Readline Support ............................... 696
7.8.2 Readline Options ....................................... 696
7.8.3 Readline Functions ..................................... 697
7.8.3.1 Readline Port ......................................... 697
7.8.3.2 Completion ........................................... 698

7.9 Pretty Printing ........................................... 699
7.10 Formatted Output ......................................... 700
7.11 File Tree Walk ............................................ 711
7.12 Queues .................................................... 716
7.13 Streams .................................................... 718
7.14 Buffered Input ............................................ 720
7.15 Expect ...................................................... 721

7.16 sxml-match: Pattern Matching of SXML .................. 723
Syntax ......................................................... 723
Matching XML Elements ....................................... 725
Ellipses in Patterns .......................................... 725
Ellipses in Quasiquote'd Output ............................... 725
Matching Node sets ........................................... 725
Matching the “Rest” of a Node set ............................. 726
Matching the Unmatched Attributes ........................... 726
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Values in Attribute Patterns</td>
<td>726</td>
</tr>
<tr>
<td>Guards in Patterns</td>
<td>726</td>
</tr>
<tr>
<td>Catamorphisms</td>
<td>727</td>
</tr>
<tr>
<td>Named-Catamorphisms</td>
<td>727</td>
</tr>
<tr>
<td>sxml-match-let and sxml-match-let*</td>
<td>728</td>
</tr>
<tr>
<td>7.17 The Scheme shell (scsh)</td>
<td>728</td>
</tr>
<tr>
<td>7.18 Curried Definitions</td>
<td>728</td>
</tr>
<tr>
<td>7.19 Statprof</td>
<td>729</td>
</tr>
<tr>
<td>7.20 SXML</td>
<td>733</td>
</tr>
<tr>
<td>7.20.1 SXML Overview</td>
<td>733</td>
</tr>
<tr>
<td>7.20.2 Reading and Writing XML</td>
<td>733</td>
</tr>
<tr>
<td>7.20.3 SSAX: A Functional XML Parsing Toolkit</td>
<td>736</td>
</tr>
<tr>
<td>7.20.3.1 History</td>
<td>736</td>
</tr>
<tr>
<td>7.20.3.2 Implementation</td>
<td>736</td>
</tr>
<tr>
<td>7.20.3.3 Usage</td>
<td>738</td>
</tr>
<tr>
<td>7.20.4 Transforming SXML</td>
<td>739</td>
</tr>
<tr>
<td>7.20.4.1 Overview</td>
<td>739</td>
</tr>
<tr>
<td>7.20.4.2 Usage</td>
<td>740</td>
</tr>
<tr>
<td>7.20.5 SXML Tree Fold</td>
<td>740</td>
</tr>
<tr>
<td>7.20.5.1 Overview</td>
<td>740</td>
</tr>
<tr>
<td>7.20.5.2 Usage</td>
<td>740</td>
</tr>
<tr>
<td>7.20.6 SXPath</td>
<td>742</td>
</tr>
<tr>
<td>7.20.6.1 Overview</td>
<td>742</td>
</tr>
<tr>
<td>7.20.6.2 Basic Converters and Applicators</td>
<td>743</td>
</tr>
<tr>
<td>7.20.6.3 Converter Combinators</td>
<td>745</td>
</tr>
<tr>
<td>7.20.7 (sxml ssax input-parse)</td>
<td>747</td>
</tr>
<tr>
<td>7.20.7.1 Overview</td>
<td>747</td>
</tr>
<tr>
<td>7.20.7.2 Usage</td>
<td>748</td>
</tr>
<tr>
<td>7.20.8 (sxml apply-templates)</td>
<td>748</td>
</tr>
<tr>
<td>7.20.8.1 Overview</td>
<td>748</td>
</tr>
<tr>
<td>7.20.8.2 Usage</td>
<td>748</td>
</tr>
<tr>
<td>7.21 Texinfo Processing</td>
<td>749</td>
</tr>
<tr>
<td>7.21.1 (texinfo)</td>
<td>749</td>
</tr>
<tr>
<td>7.21.1.1 Overview</td>
<td>749</td>
</tr>
<tr>
<td>7.21.1.2 Usage</td>
<td>749</td>
</tr>
<tr>
<td>7.21.2 (texinfo docbook)</td>
<td>750</td>
</tr>
<tr>
<td>7.21.2.1 Overview</td>
<td>750</td>
</tr>
<tr>
<td>7.21.2.2 Usage</td>
<td>750</td>
</tr>
<tr>
<td>7.21.3 (texinfo html)</td>
<td>751</td>
</tr>
<tr>
<td>7.21.3.1 Overview</td>
<td>751</td>
</tr>
<tr>
<td>7.21.3.2 Usage</td>
<td>751</td>
</tr>
<tr>
<td>7.21.4 (texinfo indexing)</td>
<td>751</td>
</tr>
<tr>
<td>7.21.4.1 Overview</td>
<td>751</td>
</tr>
<tr>
<td>7.21.4.2 Usage</td>
<td>751</td>
</tr>
<tr>
<td>7.21.5 (texinfo string-utils)</td>
<td>752</td>
</tr>
<tr>
<td>7.21.5.1 Overview</td>
<td>752</td>
</tr>
<tr>
<td>7.21.5.2 Usage</td>
<td>752</td>
</tr>
<tr>
<td>7.21.6 (texinfo plain-text)</td>
<td>755</td>
</tr>
</tbody>
</table>
9 Guile Implementation ........................................ 799

9.1 A Brief History of Guile .................................... 799
  9.1.1 The Emacs Thesis ....................................... 799
  9.1.2 Early Days ........................................... 799
  9.1.3 A Scheme of Many Maintainers ......................... 800
  9.1.4 A Timeline of Selected Guile Releases ................. 801
  9.1.5 Status, or: Your Help Needed .......................... 802

9.2 Data Representation ........................................ 803
  9.2.1 A Simple Representation ................................ 804
  9.2.2 Faster Integers ....................................... 804
  9.2.3 Cheaper Pairs ......................................... 806
  9.2.4 Conservative Garbage Collection ....................... 807
  9.2.5 The SCM Type in Guile ................................ 808
    9.2.5.1 Relationship between SCM and scm_t_bits ......... 808
    9.2.5.2 Immediate objects .................................. 809
    9.2.5.3 Non-immediate objects ............................... 810
    9.2.5.4 Allocating Cells .................................... 810
    9.2.5.5 Heap Cell Type Information ......................... 811
    9.2.5.6 Accessing Cell Entries .............................. 811

9.3 A Virtual Machine for Guile ................................ 812
  9.3.1 Why a VM? ............................................. 812
  9.3.2 VM Concepts ......................................... 813
  9.3.3 Stack Layout ......................................... 814
  9.3.4 Variables and the VM ................................ 815
  9.3.5 Compiled Procedures are VM Programs ................. 816
  9.3.6 Object File Format ................................... 817
  9.3.7 Instruction Set ....................................... 819
    9.3.7.1 Lexical Environment Instructions ................. 820
    9.3.7.2 Top-Level Environment Instructions ............... 821
    9.3.7.3 Procedure Call and Return Instructions .......... 822
    9.3.7.4 Function Prologue Instructions .................... 824
    9.3.7.5 Trampoline Instructions ............................ 825
    9.3.7.6 Branch Instructions ................................. 826
    9.3.7.7 Constant Instructions ............................... 827
    9.3.7.8 Dynamic Environment Instructions ................ 828
    9.3.7.9 Miscellaneous Instructions ......................... 829
    9.3.7.10 Inlined Scheme Instructions ....................... 830
    9.3.7.11 Inlined Atomic Instructions ....................... 832
    9.3.7.12 Inlined Mathematical Instructions ................. 832
    9.3.7.13 Inlined Bytevector Instructions .................. 833
    9.3.7.14 Unboxed Integer Arithmetic ....................... 834
    9.3.7.15 Unboxed Floating-Point Arithmetic ............... 836

9.4 Compiling to the Virtual Machine ......................... 836
  9.4.1 Compiler Tower ....................................... 836
  9.4.2 The Scheme Compiler .................................. 838
  9.4.3 Tree-IL .............................................. 839
  9.4.4 Continuation-Passing Style ............................ 843
    9.4.4.1 An Introduction to CPS ........................... 843
9.4.4.2 CPS in Guile........................................ 845
9.4.4.3 Building CPS....................................... 848
9.4.4.4 CPS Soup........................................... 849
9.4.4.5 Compiling CPS....................................... 852
9.4.5 Bytecode............................................... 853
9.4.6 Writing New High-Level Languages................... 855
9.4.7 Extending the Compiler............................... 855

Appendix A GNU Free Documentation License . 857

Concept Index .............................................. 865
Procedure Index............................................. 871
Variable Index............................................... 913
Type Index................................................... 917
R5RS Index................................................... 919
Preface

This manual describes how to use Guile, GNU’s Ubiquitous Intelligent Language for Extensions. It relates particularly to Guile version 2.2.6.

Contributors to this Manual

Like Guile itself, the Guile reference manual is a living entity, cared for by many people over a long period of time. As such, it is hard to identify individuals of whom to say “yes, this person, she wrote the manual.”

Still, among the many contributions, some caretakers stand out. First among them is Neil Jerram, who has been working on this document for ten years now. Neil’s attention both to detail and to the big picture have made a real difference in the understanding of a generation of Guile hackers.

Next we should note Marius Vollmer’s effect on this document. Marius maintained Guile during a period in which Guile’s API was clarified—put to the fire, so to speak—and he had the good sense to effect the same change on the manual.

Martin Grabmueller made substantial contributions throughout the manual in preparation for the Guile 1.6 release, including filling out a lot of the documentation of Scheme data types, control mechanisms and procedures. In addition, he wrote the documentation for Guile’s SRFI modules and modules associated with the Guile REPL.

Ludovic Courtès and Andy Wingo, the Guile maintainers at the time of this writing (late 2010), have also made their dent in the manual, writing documentation for new modules and subsystems in Guile 2.0. They are also responsible for ensuring that the existing text retains its relevance as Guile evolves. See Section 2.6 [Reporting Bugs], page 12, for more information on reporting problems in this manual.

The content for the first versions of this manual incorporated and was inspired by documents from Aubrey Jaffer, author of the SCM system on which Guile was based, and from Tom Lord, Guile’s first maintainer. Although most of this text has been rewritten, all of it was important, and some of the structure remains.

The manual for the first versions of Guile were largely written, edited, and compiled by Mark Galassi and Jim Blandy. In particular, Jim wrote the original tutorial on Guile’s data representation and the C API for accessing Guile objects.

Significant portions were also contributed by Thien-Thi Nguyen, Kevin Ryde, Mikael Djurfeldt, Christian Lynbech, Julian Graham, Gary Houston, Tim Pierce, and a few dozen more. You, reader, are most welcome to join their esteemed ranks. Visit Guile’s web site at http://www.gnu.org/software/guile/ to find out how to get involved.

The Guile License

Guile is Free Software. Guile is copyrighted, not public domain, and there are restrictions on its distribution or redistribution, but these restrictions are designed to permit everything a cooperating person would want to do.

- The Guile library (libguile) and supporting files are published under the terms of the GNU Lesser General Public License version 3 or later. See the files COPYING.LESSER and COPYING.
• The Guile readline module is published under the terms of the GNU General Public License version 3 or later. See the file COPYING.

• The manual you’re now reading is published under the terms of the GNU Free Documentation License (see Appendix A [GNU Free Documentation License], page 857).

C code linking to the Guile library is subject to terms of that library. Basically such code may be published on any terms, provided users can re-link against a new or modified version of Guile.

C code linking to the Guile readline module is subject to the terms of that module. Basically such code must be published on Free terms.

Scheme level code written to be run by Guile (but not derived from Guile itself) is not restricted in any way, and may be published on any terms. We encourage authors to publish on Free terms.

You must be aware there is no warranty whatsoever for Guile. This is described in full in the licenses.
1 Introduction

Guile is an implementation of the Scheme programming language. Scheme (http://schemers.org/) is an elegant and conceptually simple dialect of Lisp, originated by Guy Steele and Gerald Sussman, and since evolved by the series of reports known as RnRS (the Revised Reports on Scheme).

Unlike, for example, Python or Perl, Scheme has no benevolent dictator. There are many Scheme implementations, with different characteristics and with communities and academic activities around them, and the language develops as a result of the interplay between these. Guile’s particular characteristics are that

- it is easy to combine with other code written in C
- it has a historical and continuing connection with the GNU Project
- it emphasizes interactive and incremental programming
- it actually supports several languages, not just Scheme.

The next few sections explain what we mean by these points. The sections after that cover how you can obtain and install Guile, and the typographical conventions that we use in this manual.

1.1 Guile and Scheme

Guile implements Scheme as described in the Revised Report on the Algorithmic Language Scheme (usually known as R5RS), providing clean and general data and control structures. Guile goes beyond the rather austere language presented in R5RS, extending it with a module system, full access to POSIX system calls, networking support, multiple threads, dynamic linking, a foreign function call interface, powerful string processing, and many other features needed for programming in the real world.

The Scheme community has recently agreed and published R6RS, the latest installment in the RnRS series. R6RS significantly expands the core Scheme language, and standardises many non-core functions that implementations—including Guile—have previously done in different ways. Guile has been updated to incorporate some of the features of R6RS, and to adjust some existing features to conform to the R6RS specification, but it is by no means a complete R6RS implementation. See Section 7.6 [R6RS Support], page 648.

Between R5RS and R6RS, the SRFI process (http://srfi.schemers.org/) standardised interfaces for many practical needs, such as multithreaded programming and multidimensional arrays. Guile supports many SRFIs, as documented in detail in Section 7.5 [SRFI Support], page 576.

In summary, so far as relationship to the Scheme standards is concerned, Guile is an R5RS implementation with many extensions, some of which conform to SRFIs or to the relevant parts of R6RS.

1.2 Combining with C Code

Like a shell, Guile can run interactively—reading expressions from the user, evaluating them, and displaying the results—or as a script interpreter, reading and executing Scheme code from a file. Guile also provides an object library, libguile, that allows other applications
to easily incorporate a complete Scheme interpreter. An application can then use Guile as an extension language, a clean and powerful configuration language, or as multi-purpose “glue”, connecting primitives provided by the application. It is easy to call Scheme code from C code and vice versa, giving the application designer full control of how and when to invoke the interpreter. Applications can add new functions, data types, control structures, and even syntax to Guile, creating a domain-specific language tailored to the task at hand, but based on a robust language design.

This kind of combination is helped by four aspects of Guile’s design and history. First is that Guile has always been targeted as an extension language. Hence its C API has always been of great importance, and has been developed accordingly. Second and third are rather technical points—that Guile uses conservative garbage collection, and that it implements the Scheme concept of continuations by copying and reinstating the C stack—but whose practical consequence is that most existing C code can be glued into Guile as is, without needing modifications to cope with strange Scheme execution flows. Last is the module system, which helps extensions to coexist without stepping on each others’ toes.

Guile’s module system allows one to break up a large program into manageable sections with well-defined interfaces between them. Modules may contain a mixture of interpreted and compiled code; Guile can use either static or dynamic linking to incorporate compiled code. Modules also encourage developers to package up useful collections of routines for general distribution; as of this writing, one can find Emacs interfaces, database access routines, compilers, GUI toolkit interfaces, and HTTP client functions, among others.

1.3 Guile and the GNU Project

Guile was conceived by the GNU Project following the fantastic success of Emacs Lisp as an extension language within Emacs. Just as Emacs Lisp allowed complete and unanticipated applications to be written within the Emacs environment, the idea was that Guile should do the same for other GNU Project applications. This remains true today.

The idea of extensibility is closely related to the GNU project’s primary goal, that of promoting software freedom. Software freedom means that people receiving a software package can modify or enhance it to their own desires, including in ways that may not have occurred at all to the software’s original developers. For programs written in a compiled language like C, this freedom covers modifying and rebuilding the C code; but if the program also provides an extension language, that is usually a much friendlier and lower-barrier-of-entry way for the user to start making their own changes.

Guile is now used by GNU project applications such as AutoGen, Lilypond, Denemo, Mailutils, TeXmacs and GnuCash, and we hope that there will be many more in future.

1.4 Interactive Programming

Non-free software has no interest in its users being able to see how it works. They are supposed to just accept it, or to report problems and hope that the source code owners will choose to work on them.

Free software aims to work reliably just as much as non-free software does, but it should also empower its users by making its workings available. This is useful for many reasons, including education, auditing and enhancements, as well as for debugging problems.
The ideal free software system achieves this by making it easy for interested users to see
the source code for a feature that they are using, and to follow through that source code
step-by-step, as it runs. In Emacs, good examples of this are the source code hyperlinks in
the help system, and \texttt{edebug}. Then, for bonus points and maximising the ability for the
user to experiment quickly with code changes, the system should allow parts of the source
code to be modified and reloaded into the running program, to take immediate effect.

Guile is designed for this kind of interactive programming, and this distinguishes it
from many Scheme implementations that instead prioritise running a fixed Scheme program
as fast as possible—because there are tradeoffs between performance and the ability to
modify parts of an already running program. There are faster Schemes than Guile, but
Guile is a GNU project and so prioritises the GNU vision of programming freedom and
experimentation.

1.5 Supporting Multiple Languages

Since the 2.0 release, Guile’s architecture supports compiling any language to its core virtual
machine bytecode, and Scheme is just one of the supported languages. Other supported
languages are Emacs Lisp, ECMAScript (commonly known as Javascript) and Brainfuck,
and work is under discussion for Lua, Ruby and Python.

This means that users can program applications which use Guile in the language of their
choice, rather than having the tastes of the application’s author imposed on them.

1.6 Obtaining and Installing Guile

Guile can be obtained from the main GNU archive site ftp://ftp.gnu.org or any of its
mirrors. The file will be named guile-version.tar.gz. The current version is 2.2.6, so the file
you should grab is:


To unbundle Guile use the instruction

\texttt{zcat guile-2.2.6.tar.gz | tar xvf -}

which will create a directory called \texttt{guile-2.2.6} with all the sources. You can look at the
file \texttt{INSTALL} for detailed instructions on how to build and install Guile, but you should be
able to just do

\texttt{cd guile-2.2.6}

\texttt{./configure}

\texttt{make}

\texttt{make install}

This will install the Guile executable \texttt{guile}, the Guile library \texttt{libguile} and various
associated header files and support libraries. It will also install the Guile reference manual.

Since this manual frequently refers to the Scheme “standard”, also known as R5RS, or
the “Revised\textsuperscript{5} Report on the Algorithmic Language Scheme”, we have included the report in
the Guile distribution; see Section “Introduction” in Revised\textsuperscript{5} Report on the Algorithmic
Language Scheme. This will also be installed in your info directory.
1.7 Organisation of this Manual

The rest of this manual is organised into the following chapters.

Chapter 2: Hello Guile!
A whirlwind tour shows how Guile can be used interactively and as a script interpreter, how to link Guile into your own applications, and how to write modules of interpreted and compiled code for use with Guile. Everything introduced here is documented again and in full by the later parts of the manual.

Chapter 3: Hello Scheme!
For readers new to Scheme, this chapter provides an introduction to the basic ideas of the Scheme language. This material would apply to any Scheme implementation and so does not make reference to anything Guile-specific.

Chapter 4: Programming in Scheme
Provides an overview of programming in Scheme with Guile. It covers how to invoke the guile program from the command-line and how to write scripts in Scheme. It also introduces the extensions that Guile offers beyond standard Scheme.

Chapter 5: Programming in C
Provides an overview of how to use Guile in a C program. It discusses the fundamental concepts that you need to understand to access the features of Guile, such as dynamic types and the garbage collector. It explains in a tutorial like manner how to define new data types and functions for the use by Scheme programs.

Chapter 6: Guile API Reference
This part of the manual documents the Guile API in functionality-based groups with the Scheme and C interfaces presented side by side.

Chapter 7: Guile Modules
Describes some important modules, distributed as part of the Guile distribution, that extend the functionality provided by the Guile Scheme core.

Chapter 8: GOOPS
Describes GOOPS, an object oriented extension to Guile that provides classes, multiple inheritance and generic functions.

1.8 Typographical Conventions
In examples and procedure descriptions and all other places where the evaluation of Scheme expression is shown, we use some notation for denoting the output and evaluation results of expressions.

The symbol ‘⇒’ is used to tell which value is returned by an evaluation:

\[
(\texttt{(+ 1 2)}) \Rightarrow 3
\]

Some procedures produce some output besides returning a value. This is denoted by the symbol ‘⊣’.

\[
(\texttt{begin (display 1) (newline) 'hooray})
\]
\[ \downarrow 1 \quad \Rightarrow \text{hooray} \]

As you can see, this code prints ‘1’ (denoted by ‘\(\downarrow\)’), and returns \text{hooray} (denoted by ‘\(\Rightarrow\)’).
2 Hello Guile!

This chapter presents a quick tour of all the ways that Guile can be used. There are additional examples in the examples/ directory in the Guile source distribution. It also explains how best to report any problems that you find.

The following examples assume that Guile has been installed in /usr/local/.

2.1 Running Guile Interactively

In its simplest form, Guile acts as an interactive interpreter for the Scheme programming language, reading and evaluating Scheme expressions the user enters from the terminal. Here is a sample interaction between Guile and a user; the user’s input appears after the $ and scheme@(guile-user)> prompts:

```
$ guile
scheme@(guile-user)> (+ 1 2 3) ; add some numbers
$1 = 6
scheme@(guile-user)> (define (factorial n) ; define a function
    (if (zero? n) 1 (* n (factorial (- n 1)))))
scheme@(guile-user)> (factorial 20)
$2 = 2432902008176640000
scheme@(guile-user)> (getpwnam "root") ; look in /etc/passwd
$3 = #("root" "x" 0 0 "root" "/root" "/bin/bash")
scheme@(guile-user)> C-d
$
```

2.2 Running Guile Scripts

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

Here is a trivial Guile script. See Section 4.3 [Guile Scripting], page 41, for more details.

```
#!/usr/local/bin/guile -s
!
(display "Hello, world!")
(newline)
```

2.3 Linking Guile into Programs

The Guile interpreter is available as an object library, to be linked into applications using Scheme as a configuration or extension language.

Here is simple-guile.c, source code for a program that will produce a complete Guile interpreter. In addition to all usual functions provided by Guile, it will also offer the function my-hostname.

```
#include <stdlib.h>
#include <libguile.h>
```
static SCM
my_hostname (void)
{
    char *s = getenv ("HOSTNAME");
    if (s == NULL)
        return SCM_BOOL_F;
    else
        return scm_from_locale_string (s);
}

static void
inner_main (void *data, int argc, char **argv)
{
    scm_c_define_gsubr ("my-hostname", 0, 0, 0, my_hostname);
    scm_shell (argc, argv);
}

int
main (int argc, char **argv)
{
    scm_boot_guile (argc, argv, inner_main, 0);
    return 0; /* never reached */
}

When Guile is correctly installed on your system, the above program can be compiled
and linked like this:

$ gcc -o simple-guile simple-guile.c \
   'pkg-config --cflags --libs guile-2.2'

When it is run, it behaves just like the guile program except that you can also call the
new my-hostname function.

$ ./simple-guile
scheme@(guile-user)> (+ 1 2 3)
$1 = 6
scheme@(guile-user)> (my-hostname)
"burns"

2.4 Writing Guile Extensions

You can link Guile into your program and make Scheme available to the users of your
program. You can also link your library into Guile and make its functionality available to
all users of Guile.

A library that is linked into Guile is called an extension, but it really just is an ordinary
object library.

The following example shows how to write a simple extension for Guile that makes the
j0 function available to Scheme code.

#include <math.h>
#include <libguile.h>
SCM
j0_wrapper (SCM x)
{
    return scm_from_double (j0 (scm_to_double (x)));
}

void
init_bessel ()
{
    scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}

This C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux:

gcc 'pkg-config --cflags guile-2.2' \
    -shared -o libguile-bessel.so -fPIC bessel.c

For creating shared libraries portably, we recommend the use of GNU Libtool (see Section "Introduction" in GNU Libtool).

A shared library can be loaded into a running Guile process with the function load-extension. The j0 is then immediately available:

$ guile
scheme@(guile-user)> (load-extension "./libguile-bessel" "init_bessel")
scheme@(guile-user)> (j0 2)
$1 = 0.223890779141236

For more on how to install your extension, see Section 4.7 [Installing Site Packages], page 57.

2.5 Using the Guile Module System

Guile has support for dividing a program into modules. By using modules, you can group related code together and manage the composition of complete programs from largely independent parts.

For more details on the module system beyond this introductory material, See Section 6.20 [Modules], page 406.

2.5.1 Using Modules

Guile comes with a lot of useful modules, for example for string processing or command line parsing. Additionally, there exist many Guile modules written by other Guile hackers, but which have to be installed manually.

Here is a sample interactive session that shows how to use the (ice-9 popen) module which provides the means for communicating with other processes over pipes together with the (ice-9 rdelim) module that provides the function read-line.

$ guile
scheme0(guile-user)> (use-modules (ice-9 popen))
scheme0(guile-user)> (use-modules (ice-9 rdelim))
scheme0(guile-user)> (define p (open-input-pipe "ls -l"))
scheme0(guile-user)> (read-line p)
$1 = "total 30"
scheme0(guile-user)> (read-line p)
$2 = "drwxr-sr-x 2 mgrabmue mgrabmue 1024 Mar 29 19:57 CVS"
2.5.2 Writing new Modules

You can create new modules using the syntactic form `define-module`. All definitions following this form until the next `define-module` are placed into the new module.

One module is usually placed into one file, and that file is installed in a location where Guile can automatically find it. The following session shows a simple example.

```
$ cat /usr/local/share/guile/site/foo/bar.scm
(define-module (foo bar)
  #:export (frob))
(define (frob x) (* 2 x))

$ guile
scheme@(guile-user)> (use-modules (foo bar))
scheme@(guile-user)> (frob 12)
$1 = 24
```

For more on how to install your module, see Section 4.7 [Installing Site Packages], page 57.

2.5.3 Putting Extensions into Modules

In addition to Scheme code you can also put things that are defined in C into a module. You do this by writing a small Scheme file that defines the module and call `load-extension` directly in the body of the module.

```
$ cat /usr/local/share/guile/site/math/bessel.scm
(define-module (math bessel)
  #:export (j0))
(load-extension "libguile-bessel" "init_bessel")

$ file /usr/local/lib/guile/2.2/extensions/libguile-bessel.so
... ELF 32-bit LSB shared object ...

$ guile
scheme0(guile-user)> (use-modules (math bessel))
scheme0(guile-user)> (j0 2)
0.223890779141236
```

See Section 6.21.4 [Modules and Extensions], page 426, for more information.

2.6 Reporting Bugs

Any problems with the installation should be reported to bug-guile@gnu.org.

If you find a bug in Guile, please report it to the Guile developers, so they can fix it. They may also be able to suggest workarounds when it is not possible for you to apply the bug-fix or install a new version of Guile yourself.

Before sending in bug reports, please check with the following list that you really have found a bug.

- Whenever documentation and actual behavior differ, you have certainly found a bug, either in the documentation or in the program.
- When Guile crashes, it is a bug.
• When Guile hangs or takes forever to complete a task, it is a bug.
• When calculations produce wrong results, it is a bug.
• When Guile signals an error for valid Scheme programs, it is a bug.
• When Guile does not signal an error for invalid Scheme programs, it may be a bug, unless this is explicitly documented.
• When some part of the documentation is not clear and does not make sense to you even after re-reading the section, it is a bug.

Before reporting the bug, check whether any programs you have loaded into Guile, including your .guile file, set any variables that may affect the functioning of Guile. Also, see whether the problem happens in a freshly started Guile without loading your .guile file (start Guile with the -q switch to prevent loading the init file). If the problem does not occur then, you must report the precise contents of any programs that you must load into Guile in order to cause the problem to occur.

When you write a bug report, please make sure to include as much of the information described below in the report. If you can't figure out some of the items, it is not a problem, but the more information we get, the more likely we can diagnose and fix the bug.

• The version number of Guile. You can get this information from invoking `guile --version` at your shell, or calling (version) from within Guile.

• Your machine type, as determined by the config.guess shell script. If you have a Guile checkout, this file is located in build-aux; otherwise you can fetch the latest version from http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;hb=HEAD.

    $ build-aux/config.guess
    x86_64-unknown-linux-gnu

• If you installed Guile from a binary package, the version of that package. On systems that use RPM, use rpm -qa | grep guile. On systems that use DPKG, dpkg -l | grep guile.

• If you built Guile yourself, the build configuration that you used:

    $ ./config.status --config
    '--enable-error-on-warning' '--disable-deprecated'...

• A complete description of how to reproduce the bug.
    If you have a Scheme program that produces the bug, please include it in the bug report. If your program is too big to include, please try to reduce your code to a minimal test case.
    If you can reproduce your problem at the REPL, that is best. Give a transcript of the expressions you typed at the REPL.

• A description of the incorrect behavior. For example, "The Guile process gets a fatal signal," or, "The resulting output is as follows, which I think is wrong."

    If the manifestation of the bug is a Guile error message, it is important to report the precise text of the error message, and a backtrace showing how the Scheme program arrived at the error. This can be done using the ,backtrace command in Guile's debugger.
If your bug causes Guile to crash, additional information from a low-level debugger such as GDB might be helpful. If you have built Guile yourself, you can run Guile under GDB via the *meta/gdb-uninstalled-guile* script. Instead of invoking Guile as usual, invoke the wrapper script, type *run* to start the process, then *backtrace* when the crash comes. Include that backtrace in your report.
3 Hello Scheme!

In this chapter, we introduce the basic concepts that underpin the elegance and power of the Scheme language.

Readers who already possess a background knowledge of Scheme may happily skip this chapter. For the reader who is new to the language, however, the following discussions on data, procedures, expressions and closure are designed to provide a minimum level of Scheme understanding that is more or less assumed by the chapters that follow.

The style of this introductory material aims about halfway between the terse precision of R5RS and the discursiveness of existing Scheme tutorials. For pointers to useful Scheme resources on the web, please see Section 3.5 [Further Reading], page 34.

3.1 Data Types, Values and Variables

This section discusses the representation of data types and values, what it means for Scheme to be a latently typed language, and the role of variables. We conclude by introducing the Scheme syntaxes for defining a new variable, and for changing the value of an existing variable.

3.1.1 Latent Typing

The term latent typing is used to describe a computer language, such as Scheme, for which you cannot, in general, simply look at a program’s source code and determine what type of data will be associated with a particular variable, or with the result of a particular expression.

Sometimes, of course, you can tell from the code what the type of an expression will be. If you have a line in your program that sets the variable x to the numeric value 1, you can be certain that, immediately after that line has executed (and in the absence of multiple threads), x has the numeric value 1. Or if you write a procedure that is designed to concatenate two strings, it is likely that the rest of your application will always invoke this procedure with two string parameters, and quite probable that the procedure would go wrong in some way if it was ever invoked with parameters that were not both strings.

Nevertheless, the point is that there is nothing in Scheme which requires the procedure parameters always to be strings, or x always to hold a numeric value, and there is no way of declaring in your program that such constraints should always be obeyed. In the same vein, there is no way to declare the expected type of a procedure’s return value.

Instead, the types of variables and expressions are only known – in general – at run time. If you need to check at some point that a value has the expected type, Scheme provides run time procedures that you can invoke to do so. But equally, it can be perfectly valid for two separate invocations of the same procedure to specify arguments with different types, and to return values with different types.

The next subsection explains what this means in practice, for the ways that Scheme programs use data types, values and variables.

3.1.2 Values and Variables

Scheme provides many data types that you can use to represent your data. Primitive types include characters, strings, numbers and procedures. Compound types, which allow a group
of primitive and compound values to be stored together, include lists, pairs, vectors and multi-dimensional arrays. In addition, Guile allows applications to define their own data types, with the same status as the built-in standard Scheme types.

As a Scheme program runs, values of all types pop in and out of existence. Sometimes values are stored in variables, but more commonly they pass seamlessly from being the result of one computation to being one of the parameters for the next.

Consider an example. A string value is created because the interpreter reads in a literal string from your program’s source code. Then a numeric value is created as the result of calculating the length of the string. A second numeric value is created by doubling the calculated length. Finally the program creates a list with two elements – the doubled length and the original string itself – and stores this list in a program variable.

All of the values involved here – in fact, all values in Scheme – carry their type with them. In other words, every value “knows,” at runtime, what kind of value it is. A number, a string, a list, whatever.

A variable, on the other hand, has no fixed type. A variable – \textit{x}, say – is simply the name of a location – a box – in which you can store any kind of Scheme value. So the same variable in a program may hold a number at one moment, a list of procedures the next, and later a pair of strings. The “type” of a variable – insofar as the idea is meaningful at all – is simply the type of whatever value the variable happens to be storing at a particular moment.

### 3.1.3 Defining and Setting Variables

To define a new variable, you use Scheme’s \texttt{define} syntax like this:

\[
\text{(define variable-name value)}
\]

This makes a new variable called \textit{variable-name} and stores \textit{value} in it as the variable’s initial value. For example:

\[
\begin{align*}
\text{;; Make a variable ‘x’ with initial numeric value 1.} \\
\text{(define x 1)}
\end{align*}
\]

\[
\begin{align*}
\text{;; Make a variable ‘organization’ with an initial string value.} \\
\text{(define organization "Free Software Foundation")}
\end{align*}
\]

(In Scheme, a semicolon marks the beginning of a comment that continues until the end of the line. So the lines beginning ;; are comments.)

Changing the value of an already existing variable is very similar, except that \texttt{define} is replaced by the Scheme syntax \texttt{set!}, like this:

\[
\text{(set! variable-name new-value)}
\]

Remember that variables do not have fixed types, so \textit{new-value} may have a completely different type from whatever was previously stored in the location named by \textit{variable-name}. Both of the following examples are therefore correct.

\[
\begin{align*}
\text{;; Change the value of ‘x’ to 5.} \\
\text{(set! x 5)}
\end{align*}
\]

\[
\begin{align*}
\text{;; Change the value of ‘organization’ to the FSF’s street number.} \\
\text{(set! organization 545)}
\end{align*}
\]
In these examples, \textit{value} and \textit{new-value} are literal numeric or string values. In general, however, \textit{value} and \textit{new-value} can be any Scheme expression. Even though we have not yet covered the forms that Scheme expressions can take (see Section 3.3 [About Expressions], page 20), you can probably guess what the following \texttt{set!} example does...

\begin{verbatim}
(set! x (+ x 1))
\end{verbatim}

(Note: this is not a complete description of \texttt{define} and \texttt{set!}, because we need to introduce some other aspects of Scheme before the missing pieces can be filled in. If, however, you are already familiar with the structure of Scheme, you may like to read about those missing pieces immediately by jumping ahead to the following references.

- Section 3.2.4 [Lambda Alternatives], page 20, to read about an alternative form of the \texttt{define} syntax that can be used when defining new procedures.
- Section 6.9.8 [Procedures with Setters], page 259, to read about an alternative form of the \texttt{set!} syntax that helps with changing a single value in the depths of a compound data structure.)
- See Section 6.12.3 [Internal Definitions], page 296, to read about using \texttt{define} other than at top level in a Scheme program, including a discussion of when it works to use \texttt{define} rather than \texttt{set!} to change the value of an existing variable.

### 3.2 The Representation and Use of Procedures

This section introduces the basics of using and creating Scheme procedures. It discusses the representation of procedures as just another kind of Scheme value, and shows how procedure invocation expressions are constructed. We then explain how \texttt{lambda} is used to create new procedures, and conclude by presenting the various shorthand forms of \texttt{define} that can be used instead of writing an explicit \texttt{lambda} expression.

#### 3.2.1 Procedures as Values

One of the great simplifications of Scheme is that a procedure is just another type of value, and that procedure values can be passed around and stored in variables in exactly the same way as, for example, strings and lists. When we talk about a built-in standard Scheme procedure such as \texttt{open-input-file}, what we actually mean is that there is a pre-defined top level variable called \texttt{open-input-file}, whose value is a procedure that implements what R5RS says that \texttt{open-input-file} should do.

Note that this is quite different from many dialects of Lisp — including Emacs Lisp — in which a program can use the same name with two quite separate meanings: one meaning identifies a Lisp function, while the other meaning identifies a Lisp variable, whose value need have nothing to do with the function that is associated with the first meaning. In these dialects, functions and variables are said to live in different \texttt{namespaces}.

In Scheme, on the other hand, all names belong to a single unified namespace, and the variables that these names identify can hold any kind of Scheme value, including procedure values.

One consequence of the “procedures as values” idea is that, if you don’t happen to like the standard name for a Scheme procedure, you can change it.

For example, \texttt{call-with-current-continuation} is a very important standard Scheme procedure, but it also has a very long name! So, many programmers use the following definition to assign the same procedure value to the more convenient name \texttt{call/cc}.
(define call/cc call-with-current-continuation)

Let’s understand exactly how this works. The definition creates a new variable call/cc, and then sets its value to the value of the variable call-with-current-continuation; the latter value is a procedure that implements the behaviour that R5RS specifies under the name “call-with-current-continuation”. So call/cc ends up holding this value as well.

Now that call/cc holds the required procedure value, you could choose to use call-with-current-continuation for a completely different purpose, or just change its value so that you will get an error if you accidentally use call-with-current-continuation as a procedure in your program rather than call/cc. For example:

(set! call-with-current-continuation "Not a procedure any more!")

Or you could just leave call-with-current-continuation as it was. It’s perfectly fine for more than one variable to hold the same procedure value.

### 3.2.2 Simple Procedure Invocation

A procedure invocation in Scheme is written like this:

```
(procedure [arg1 [arg2 [...]]])
```

In this expression, procedure can be any Scheme expression whose value is a procedure. Most commonly, however, procedure is simply the name of a variable whose value is a procedure.

For example, `string-append` is a standard Scheme procedure whose behaviour is to concatenate together all the arguments, which are expected to be strings, that it is given. So the expression

```
(string-append "/home" "/" "andrew")
```

is a procedure invocation whose result is the string value "/home/andrew".

Similarly, `string-length` is a standard Scheme procedure that returns the length of a single string argument, so

```
(string-length "abc")
```

is a procedure invocation whose result is the numeric value 3.

Each of the parameters in a procedure invocation can itself be any Scheme expression. Since a procedure invocation is itself a type of expression, we can put these two examples together to get

```
(string-length (string-append "/home" "/" "andrew"))
```

—a procedure invocation whose result is the numeric value 12.

(You may be wondering what happens if the two examples are combined the other way round. If we do this, we can make a procedure invocation expression that is syntactically correct:

```
(string-append "/home" (string-length "abc"))
```

but when this expression is executed, it will cause an error, because the result of (string-length "abc") is a numeric value, and string-append is not designed to accept a numeric value as one of its arguments.)
3.2.3 Creating and Using a New Procedure

Scheme has lots of standard procedures, and Guile provides all of these via predefined top level variables. All of these standard procedures are documented in the later chapters of this reference manual.

Before very long, though, you will want to create new procedures that encapsulate aspects of your own applications’ functionality. To do this, you can use the famous \texttt{lambda} syntax.

For example, the value of the following Scheme expression

\[
\text{(lambda (name address) expression ...)}
\]

is a newly created procedure that takes two arguments: \texttt{name} and \texttt{address}. The behaviour of the new procedure is determined by the sequence of \texttt{expression}s in the \textit{body} of the procedure definition. (Typically, these \texttt{expression}s would use the arguments in some way, or else there wouldn’t be any point in giving them to the procedure.) When invoked, the new procedure returns a value that is the value of the last \texttt{expression} in the procedure body.

To make things more concrete, let’s suppose that the two arguments are both strings, and that the purpose of this procedure is to form a combined string that includes these arguments. Then the full \texttt{lambda} expression might look like this:

\[
\text{(lambda (name address) (string-append "Name=" name ":Address=" address))}
\]

We noted in the previous subsection that the \texttt{procedure} part of a procedure invocation expression can be any Scheme expression whose value is a procedure. But that’s exactly what a \texttt{lambda} expression is! So we can use a \texttt{lambda} expression directly in a procedure invocation, like this:

\[
\text{((lambda (name address) (string-append "Name=" name ":Address=" address)) "FSF" "Cambridge")}
\]

This is a valid procedure invocation expression, and its result is the string:

"Name=FSF:Address=Cambridge"

It is more common, though, to store the procedure value in a variable —

\[
\text{(define make-combined-string (lambda (name address) (string-append "Name=" name ":Address=" address)))}
\]

— and then to use the variable name in the procedure invocation:

\[
\text{(make-combined-string "FSF" "Cambridge")}
\]

Which has exactly the same result.

It’s important to note that procedures created using \texttt{lambda} have exactly the same status as the standard built in Scheme procedures, and can be invoked, passed around, and stored in variables in exactly the same ways.
3.2.4 Lambda Alternatives

Since it is so common in Scheme programs to want to create a procedure and then store it in a variable, there is an alternative form of the define syntax that allows you to do just that.

A define expression of the form

```
(define (name [arg1 [arg2 ...]])
  expression ...)
```

is exactly equivalent to the longer form

```
(define name
  (lambda ([arg1 [arg2 ...]])
    expression ...))
```

So, for example, the definition of `make-combined-string` in the previous subsection could equally be written:

```
(define (make-combined-string name address)
  (string-append "Name=" name ":Address=" address))
```

This kind of procedure definition creates a procedure that requires exactly the expected number of arguments. There are two further forms of the `lambda` expression, which create a procedure that can accept a variable number of arguments:

```
(lambda (arg1 .... args) expression ...)
```

```
(lambda args expression ...)
```

The corresponding forms of the alternative define syntax are:

```
(define (name arg1 .... args) expression ...)
```

```
(define (name . args) expression ...)
```

For details on how these forms work, see See Section 6.9.1 [Lambda], page 248.

Prior to Guile 2.0, Guile provided an extension to define syntax that allowed you to nest the previous extension up to an arbitrary depth. These are no longer provided by default, and instead have been moved to Section 7.18 [Curried Definitions], page 728.

(It could be argued that the alternative define forms are rather confusing, especially for newcomers to the Scheme language, as they hide both the role of lambda and the fact that procedures are values that are stored in variables in the same way as any other kind of value. On the other hand, they are very convenient, and they are also a good example of another of Scheme’s powerful features: the ability to specify arbitrary syntactic transformations at run time, which can be applied to subsequently read input.)

3.3 Expressions and Evaluation

So far, we have met expressions that do things, such as the define expressions that create and initialize new variables, and we have also talked about expressions that have values, for example the value of the procedure invocation expression:

```
(string-append "/home" "/" "andrew")
```
but we haven’t yet been precise about what causes an expression like this procedure invocation to be reduced to its “value”, or how the processing of such expressions relates to the execution of a Scheme program as a whole.

This section clarifies what we mean by an expression’s value, by introducing the idea of evaluation. It discusses the side effects that evaluation can have, explains how each of the various types of Scheme expression is evaluated, and describes the behaviour and use of the Guile REPL as a mechanism for exploring evaluation. The section concludes with a very brief summary of Scheme’s common syntactic expressions.

3.3.1 Evaluating Expressions and Executing Programs

In Scheme, the process of executing an expression is known as evaluation. Evaluation has two kinds of result:

- the value of the evaluated expression
- the side effects of the evaluation, which consist of any effects of evaluating the expression that are not represented by the value.

Of the expressions that we have met so far, define and set! expressions have side effects — the creation or modification of a variable — but no value; lambda expressions have values — the newly constructed procedures — but no side effects; and procedure invocation expressions, in general, have either values, or side effects, or both.

It is tempting to try to define more intuitively what we mean by “value” and “side effects”, and what the difference between them is. In general, though, this is extremely difficult. It is also unnecessary; instead, we can quite happily define the behaviour of a Scheme program by specifying how Scheme executes a program as a whole, and then by describing the value and side effects of evaluation for each type of expression individually.

So, some definitions...  

- A Scheme program consists of a sequence of expressions.
- A Scheme interpreter executes the program by evaluating these expressions in order, one by one.
- An expression can be  
  - a piece of literal data, such as a number 2.3 or a string "Hello world!"
  - a variable name
  - a procedure invocation expression
  - one of Scheme’s special syntactic expressions.

The following subsections describe how each of these types of expression is evaluated.

3.3.1.1 Evaluating Literal Data

When a literal data expression is evaluated, the value of the expression is simply the value that the expression describes. The evaluation of a literal data expression has no side effects. So, for example,

- the value of the expression "abc" is the string value "abc"

---

1 These definitions are approximate. For the whole and detailed truth, see Section “Formal syntax and semantics” in The Revised(5) Report on the Algorithmic Language Scheme.
the value of the expression $3+4i$ is the complex number $3 + 4i$

the value of the expression #$\{1\ 2\ 3\}$ is a three-element vector containing the numeric values $1$, $2$ and $3$.

For any data type which can be expressed literally like this, the syntax of the literal data expression for that data type — in other words, what you need to write in your code to indicate a literal value of that type — is known as the data type’s read syntax. This manual specifies the read syntax for each such data type in the section that describes that data type.

Some data types do not have a read syntax. Procedures, for example, cannot be expressed as literal data; they must be created using a lambda expression (see Section 3.2.3 [Creating a Procedure], page 19) or implicitly using the shorthand form of define (see Section 3.2.4 [Lambda Alternatives], page 20).

### 3.3.1.2 Evaluating a Variable Reference

When an expression that consists simply of a variable name is evaluated, the value of the expression is the value of the named variable. The evaluation of a variable reference expression has no side effects.

So, after

```
(define key "Paul Evans")
```

the value of the expression `key` is the string value "Paul Evans". If `key` is then modified by

```
(set! key 3.74)
```

the value of the expression `key` is the numeric value $3.74$.

If there is no variable with the specified name, evaluation of the variable reference expression signals an error.

### 3.3.1.3 Evaluating a Procedure Invocation Expression

This is where evaluation starts getting interesting! As already noted, a procedure invocation expression has the form

```
(procedure [arg1 [arg2 ...]])
```

where `procedure` must be an expression whose value, when evaluated, is a procedure.

The evaluation of a procedure invocation expression like this proceeds by

- evaluating individually the expressions `procedure`, `arg1`, `arg2`, and so on
- calling the procedure that is the value of the `procedure` expression with the list of values obtained from the evaluations of `arg1`, `arg2` etc. as its parameters.

For a procedure defined in Scheme, “calling the procedure with the list of values as its parameters” means binding the values to the procedure’s formal parameters and then evaluating the sequence of expressions that make up the body of the procedure definition. The value of the procedure invocation expression is the value of the last evaluated expression in the procedure body. The side effects of calling the procedure are the combination of the side effects of the sequence of evaluations of expressions in the procedure body.

For a built-in procedure, the value and side-effects of calling the procedure are best described by that procedure’s documentation.
Note that the complete side effects of evaluating a procedure invocation expression consist not only of the side effects of the procedure call, but also of any side effects of the preceding evaluation of the expressions procedure, arg1, arg2, and so on.

To illustrate this, let’s look again at the procedure invocation expression:

\[(\text{string-length} (\text{string-append} "\text{/home}" "/" "\text{andrew}"))\]

In the outermost expression, \text{procedure} is \text{string-length} and \text{arg1} is \((\text{string-append} "\text{/home}" "/" "\text{andrew}"))

- Evaluation of \text{string-length}, which is a variable, gives a procedure value that implements the expected behaviour for “string-length”.
- Evaluation of \((\text{string-append} "\text{/home}" "/" "\text{andrew}"))\), which is another procedure invocation expression, means evaluating each of
  - \text{string-append}, which gives a procedure value that implements the expected behaviour for “string-append”
  - "/home", which gives the string value "/home"
  - "/", which gives the string value "/"
  - "andrew", which gives the string value "andrew"

and then invoking the procedure value with this list of string values as its arguments. The resulting value is a single string value that is the concatenation of all the arguments, namely "/home/andrew".

In the evaluation of the outermost expression, the interpreter can now invoke the procedure value obtained from \text{procedure} with the value obtained from \text{arg1} as its arguments. The resulting value is a numeric value that is the length of the argument string, which is 12.

3.3.1.4 Evaluating Special Syntactic Expressions

When a procedure invocation expression is evaluated, the procedure and \textit{all} the argument expressions must be evaluated before the procedure can be invoked. Special syntactic expressions are special because they are able to manipulate their arguments in an unevaluated form, and can choose whether to evaluate any or all of the argument expressions.

Why is this needed? Consider a program fragment that asks the user whether or not to delete a file, and then deletes the file if the user answers yes.

\[(\text{if} (\text{string=?} (\text{read-answer} "Should I delete this file?") "yes")
\text{(delete-file file)})\]

If the outermost \text{if} expression here was a procedure invocation expression, the expression \((\text{delete-file file})\), whose side effect is to actually delete a file, would already have been evaluated before the \text{if} procedure even got invoked! Clearly this is no use — the whole point of an \text{if} expression is that the \textit{consequent} expression is only evaluated if the condition of the \text{if} expression is “true”.

Therefore \text{if} must be special syntax, not a procedure. Other special syntaxes that we have already met are \texttt{define}, \texttt{set!} and \texttt{lambda}. \texttt{define} and \texttt{set!} are syntax because they need to know the variable \textit{name} that is given as the first argument in a \texttt{define} or \texttt{set!} expression, not that variable’s value. \texttt{lambda} is syntax because it does not immediately
evaluate the expressions that define the procedure body; instead it creates a procedure object that incorporates these expressions so that they can be evaluated in the future, when that procedure is invoked.

The rules for evaluating each special syntactic expression are specified individually for each special syntax. For a summary of standard special syntax, see See Section 3.3.4 [Syntax Summary], page 25.

### 3.3.2 Tail calls

Scheme is “properly tail recursive”, meaning that tail calls or recursions from certain contexts do not consume stack space or other resources and can therefore be used on arbitrarily large data or for an arbitrarily long calculation. Consider for example,

```scheme
(define (foo n)
  (display n)
  (newline)
  (foo (1+ n)))

(foo 1)
⇒ 1
2
3
...

foo prints numbers infinitely, starting from the given n. It’s implemented by printing n then recursing to itself to print n + 1 and so on. This recursion is a tail call, it’s the last thing done, and in Scheme such tail calls can be made without limit.

Or consider a case where a value is returned, a version of the SRFI-1 \texttt{last} function (see Section 7.5.3.3 [SRFI-1 Selectors], page 580) returning the last element of a list,

```scheme
(define (my-last lst)
  (if (null? (cdr lst))
      (car lst)
      (my-last (cdr lst))))

(my-last '(1 2 3)) ⇒ 3
```

If the list has more than one element, \texttt{my-last} applies itself to the \texttt{cdr}. This recursion is a tail call, there’s no code after it, and the return value is the return value from that call. In Scheme this can be used on an arbitrarily long list argument.

A proper tail call is only available from certain contexts, namely the following special form positions,

- \texttt{and} — last expression
- \texttt{begin} — last expression
- \texttt{case} — last expression in each clause
- \texttt{cond} — last expression in each clause, and the call to a \texttt{=} procedure is a tail call
- \texttt{do} — last result expression
• if — “true” and “false” leg expressions
• lambda — last expression in body
• let, let*, letrec, let-syntax, letrec-syntax — last expression in body
• or — last expression

The following core functions make tail calls,
• apply — tail call to given procedure
• call-with-current-continuation — tail call to the procedure receiving the new continuation
• call-with-values — tail call to the values-receiving procedure
• eval — tail call to evaluate the form
• string-any, string-every — tail call to predicate on the last character (if that point is reached)

The above are just core functions and special forms. Tail calls in other modules are described with the relevant documentation, for example SRFI-1 any and every (see Section 7.5.3.7 [SRFI-1 Searching], page 586).

It will be noted there are a lot of places which could potentially be tail calls, for instance the last call in a for-each, but only those explicitly described are guaranteed.

### 3.3.3 Using the Guile REPL

If you start Guile without specifying a particular program for it to execute, Guile enters its standard Read Evaluate Print Loop — or REPL for short. In this mode, Guile repeatedly reads in the next Scheme expression that the user types, evaluates it, and prints the resulting value.

The REPL is a useful mechanism for exploring the evaluation behaviour described in the previous subsection. If you type string-append, for example, the REPL replies #<primitive-procedure string-append>, illustrating the relationship between the variable string-append and the procedure value stored in that variable.

In this manual, the notation ⇒ is used to mean “evaluates to”. Wherever you see an example of the form

```
expression
⇒
result
```

feel free to try it out yourself by typing expression into the REPL and checking that it gives the expected result.

### 3.3.4 Summary of Common Syntax

This subsection lists the most commonly used Scheme syntactic expressions, simply so that you will recognize common special syntax when you see it. For a full description of each of these syntaxes, follow the appropriate reference.

lambda (see Section 6.9.1 [Lambda], page 248) is used to construct procedure objects.

define (see Section 6.12.1 [Top Level], page 293) is used to create a new variable and set its initial value.
set! (see Section 6.12.1 [Top Level], page 293) is used to modify an existing variable’s value.

let, let* and letrec (see Section 6.12.2 [Local Bindings], page 295) create an inner lexical environment for the evaluation of a sequence of expressions, in which a specified set of local variables is bound to the values of a corresponding set of expressions. For an introduction to environments, see See Section 3.4 [About Closure], page 26.

begin (see Section 6.13.1 [begin], page 298) executes a sequence of expressions in order and returns the value of the last expression. Note that this is not the same as a procedure which returns its last argument, because the evaluation of a procedure invocation expression does not guarantee to evaluate the arguments in order.

if and cond (see Section 6.13.2 [Conditionals], page 299) provide conditional evaluation of argument expressions depending on whether one or more conditions evaluate to “true” or “false”.

case (see Section 6.13.2 [Conditionals], page 299) provides conditional evaluation of argument expressions depending on whether a variable has one of a specified group of values.

and (see Section 6.13.3 [and or], page 301) executes a sequence of expressions in order until either there are no expressions left, or one of them evaluates to “false”.

or (see Section 6.13.3 [and or], page 301) executes a sequence of expressions in order until either there are no expressions left, or one of them evaluates to “true”.

3.4 The Concept of Closure

The concept of closure is the idea that a lambda expression “captures” the variable bindings that are in lexical scope at the point where the lambda expression occurs. The procedure created by the lambda expression can refer to and mutate the captured bindings, and the values of those bindings persist between procedure calls.

This section explains and explores the various parts of this idea in more detail.

3.4.1 Names, Locations, Values and Environments

We said earlier that a variable name in a Scheme program is associated with a location in which any kind of Scheme value may be stored. (Incidentally, the term “vcell” is often used in Lisp and Scheme circles as an alternative to “location”.) Thus part of what we mean when we talk about “creating a variable” is in fact establishing an association between a name, or identifier, that is used by the Scheme program code, and the variable location to which that name refers. Although the value that is stored in that location may change, the location to which a given name refers is always the same.

We can illustrate this by breaking down the operation of the define syntax into three parts: define

- creates a new location
- establishes an association between that location and the name specified as the first argument of the define expression
- stores in that location the value obtained by evaluating the second argument of the define expression.
A collection of associations between names and locations is called an environment. When you create a top level variable in a program using `define`, the name-location association for that variable is added to the “top level” environment. The “top level” environment also includes name-location associations for all the procedures that are supplied by standard Scheme.

It is also possible to create environments other than the top level one, and to create variable bindings, or name-location associations, in those environments. This ability is a key ingredient in the concept of closure; the next subsection shows how it is done.

### 3.4.2 Local Variables and Environments

We have seen how to create top level variables using the `define` syntax (see Section 3.1.3 [Definition], page 16). It is often useful to create variables that are more limited in their scope, typically as part of a procedure body. In Scheme, this is done using the `let` syntax, or one of its modified forms `let*` and `letrec`. These syntaxes are described in full later in the manual (see Section 6.12.2 [Local Bindings], page 295). Here our purpose is to illustrate their use just enough that we can see how local variables work.

For example, the following code uses a local variable `s` to simplify the computation of the area of a triangle given the lengths of its three sides.

```scheme
(define a 5.3)
(define b 4.7)
(define c 2.8)

(let ((s (/ (+ a b c) 2)))
  (sqrt (* s (- s a) (- s b) (- s c)))))
```

The effect of the `let` expression is to create a new environment and, within this environment, an association between the name `s` and a new location whose initial value is obtained by evaluating `(/ (+ a b c) 2)`. The expressions in the body of the `let`, namely `(sqrt (* s (- s a) (- s b) (- s c)))`, are then evaluated in the context of the new environment, and the value of the last expression evaluated becomes the value of the whole `let` expression, and therefore the value of the variable `area`.

### 3.4.3 Environment Chaining

In the example of the previous subsection, we glossed over an important point. The body of the `let` expression in that example refers not only to the local variable `s`, but also to the top level variables `a`, `b`, `c` and `sqrt`. (`sqrt` is the standard Scheme procedure for calculating a square root.) If the body of the `let` expression is evaluated in the context of the local `let` environment, how does the evaluation get at the values of these top level variables?

The answer is that the local environment created by a `let` expression automatically has a reference to its containing environment — in this case the top level environment — and that the Scheme interpreter automatically looks for a variable binding in the containing environment if it doesn’t find one in the local environment. More generally, every environment except for the top level one has a reference to its containing environment, and the interpreter keeps searching back up the chain of environments — from most local to top level — until it either finds a variable binding for the required identifier or exhausts the chain.
This description also determines what happens when there is more than one variable binding with the same name. Suppose, continuing the example of the previous subsection, that there was also a pre-existing top level variable \texttt{s} created by the expression:

\begin{verbatim}
(define s "Some beans, my lord!"
\end{verbatim}

Then both the top level environment and the local \texttt{let} environment would contain bindings for the name \texttt{s}. When evaluating code within the \texttt{let} body, the interpreter looks first in the local \texttt{let} environment, and so finds the binding for \texttt{s} created by the \texttt{let} syntax. Even though this environment has a reference to the top level environment, which also has a binding for \texttt{s}, the interpreter doesn't get as far as looking there. When evaluating code outside the \texttt{let} body, the interpreter looks up variable names in the top level environment, so the name \texttt{s} refers to the top level variable.

Within the \texttt{let} body, the binding for \texttt{s} in the local environment is said to shadow the binding for \texttt{s} in the top level environment.

### 3.4.4 Lexical Scope

The rules that we have just been describing are the details of how Scheme implements "lexical scoping". This subsection takes a brief diversion to explain what lexical scope means in general and to present an example of non-lexical scoping.

"Lexical scope" in general is the idea that

- an identifier at a particular place in a program always refers to the same variable location — where "always" means "every time that the containing expression is executed", and that
- the variable location to which it refers can be determined by static examination of the source code context in which that identifier appears, without having to consider the flow of execution through the program as a whole.

In practice, lexical scoping is the norm for most programming languages, and probably corresponds to what you would intuitively consider to be "normal". You may even be wondering how the situation could possibly — and usefully — be otherwise. To demonstrate that another kind of scoping is possible, therefore, and to compare it against lexical scoping, the following subsection presents an example of non-lexical scoping and examines in detail how its behavior differs from the corresponding lexically scoped code.

### 3.4.4.1 An Example of Non-Lexical Scoping

To demonstrate that non-lexical scoping does exist and can be useful, we present the following example from Emacs Lisp, which is a "dynamically scoped" language.

\begin{verbatim}
(defvar currency-abbreviation "USD")

(defun currency-string (units hundredths)
  (concat currency-abbreviation
          (number-to-string units)
          "."
          (number-to-string hundredths)))

(defun french-currency-string (units hundredths)
  (let ((currency-abbreviation "FRF"))
    (currency-string units hundredths)))
\end{verbatim}
The question to focus on here is: what does the identifier currency-abbreviation refer to in the currency-string function? The answer, in Emacs Lisp, is that all variable bindings go onto a single stack, and that currency-abbreviation refers to the topmost binding from that stack which has the name “currency-abbreviation”. The binding that is created by the defvar form, to the value "USD", is only relevant if none of the code that calls currency-string rebinds the name “currency-abbreviation” in the meanwhile.

The second function french-currency-string works precisely by taking advantage of this behaviour. It creates a new binding for the name “currency-abbreviation” which overrides the one established by the defvar form.

;; Note! This is Emacs Lisp evaluation, not Scheme!
(french-currency-string 33 44)
⇒
"FRF33.44"

Now let’s look at the corresponding, lexically scoped Scheme code:

(define currency-abbreviation "USD")

(define (currency-string units hundredths))
  (string-append currency-abbreviation
                   (number->string units)
                   
                   (number->string hundredths)))

(define (french-currency-string units hundredths)
  (let ((currency-abbreviation "FRF"))
    (currency-string units hundredths)))

According to the rules of lexical scoping, the currency-abbreviation in currency-string refers to the variable location in the innermost environment at that point in the code which has a binding for currency-abbreviation, which is the variable location in the top level environment created by the preceding (define currency-abbreviation ...) expression.

In Scheme, therefore, the french-currency-string procedure does not work as intended. The variable binding that it creates for “currency-abbreviation” is purely local to the code that forms the body of the let expression. Since this code doesn’t directly use the name “currency-abbreviation” at all, the binding is pointless.

(french-currency-string 33 44)
⇒
"USD33.44"

This begs the question of how the Emacs Lisp behaviour can be implemented in Scheme. In general, this is a design question whose answer depends upon the problem that is being addressed. In this case, the best answer may be that currency-string should be redesigned so that it can take an optional third argument. This third argument, if supplied, is interpreted as a currency abbreviation that overrides the default.
It is possible to change \texttt{french-currency-string} so that it mostly works without changing \texttt{currency-string}, but the fix is inelegant, and susceptible to interrupts that could leave the \texttt{currency-abbreviation} variable in the wrong state:

\begin{verbatim}
(define (french-currency-string units hundredths)
  (set! currency-abbreviation "FRF")
  (let ((result (currency-string units hundredths)))
    (set! currency-abbreviation "USD")
    result))
\end{verbatim}

The key point here is that the code does not create any local binding for the identifier \texttt{currency-abbreviation}, so all occurrences of this identifier refer to the top level variable.

### 3.4.5 Closure

Consider a \texttt{let} expression that doesn’t contain any \texttt{lambda}s:

\begin{verbatim}
(let ((s (/ (+ a b c) 2)))
  (sqrt (* s (- s a) (- s b) (- s c))))
\end{verbatim}

When the Scheme interpreter evaluates this, it

- creates a new environment with a reference to the environment that was current when it encountered the \texttt{let}
- creates a variable binding for \texttt{s} in the new environment, with value given by \((/ (+ a b c) 2)\)
- evaluates the expression in the body of the \texttt{let} in the context of the new local environment, and remembers the value \(V\)
- forgets the local environment
- continues evaluating the expression that contained the \texttt{let}, using the value \(V\) as the value of the \texttt{let} expression, in the context of the containing environment.

After the \texttt{let} expression has been evaluated, the local environment that was created is simply forgotten, and there is no longer any way to access the binding that was created in this environment. If the same code is evaluated again, it will follow the same steps again, creating a second new local environment that has no connection with the first, and then forgetting this one as well.

If the \texttt{let} body contains a \texttt{lambda} expression, however, the local environment is \emph{not} forgotten. Instead, it becomes associated with the procedure that is created by the \texttt{lambda} expression, and is reinstated every time that that procedure is called. In detail, this works as follows.

- When the Scheme interpreter evaluates a \texttt{lambda} expression, to create a procedure object, it stores the current environment as part of the procedure definition.
- Then, whenever that procedure is called, the interpreter reinstates the environment that is stored in the procedure definition and evaluates the procedure body within the context of that environment.

The result is that the procedure body is always evaluated in the context of the environment that was current when the procedure was created.

This is what is meant by closure. The next few subsections present examples that explore the usefulness of this concept.
3.4.6 Example 1: A Serial Number Generator

This example uses closure to create a procedure with a variable binding that is private to the procedure, like a local variable, but whose value persists between procedure calls.

```
(define (make-serial-number-generator)
  (let ((current-serial-number 0))
    (lambda ()
      (set! current-serial-number (+ current-serial-number 1))
      current-serial-number)))
```

```
(define entry-sn-generator (make-serial-number-generator))

(entry-sn-generator)
⇒
1

(entry-sn-generator)
⇒
2
```

When `make-serial-number-generator` is called, it creates a local environment with a binding for `current-serial-number` whose initial value is 0, then, within this environment, creates a procedure. The local environment is stored within the created procedure object and so persists for the lifetime of the created procedure.

Every time the created procedure is invoked, it increments the value of the `current-serial-number` binding in the captured environment and then returns the current value.

Note that `make-serial-number-generator` can be called again to create a second serial number generator that is independent of the first. Every new invocation of `make-serial-number-generator` creates a new local `let` environment and returns a new procedure object with an association to this environment.

3.4.7 Example 2: A Shared Persistent Variable

This example uses closure to create two procedures, `get-balance` and `deposit`, that both refer to the same captured local environment so that they can both access the `balance` variable binding inside that environment. The value of this variable binding persists between calls to either procedure.

Note that the captured `balance` variable binding is private to these two procedures: it is not directly accessible to any other code. It can only be accessed indirectly via `get-balance` or `deposit`, as illustrated by the `withdraw` procedure.

```
(define get-balance #f)
(define deposit #f)

(let ((balance 0))
  (set! get-balance
    (lambda ()
      balance)))
```
(set! deposit
  (lambda (amount)
    (set! balance (+ balance amount))
    balance)))

(define (withdraw amount)
  (deposit (- amount)))

(get-balance)
⇒
0

(deposit 50)
⇒
50

(withdraw 75)
⇒
-25

An important detail here is that the get-balance and deposit variables must be set up by defineing them at top level and then set!ing their values inside the let body. Using define within the let body would not work: this would create variable bindings within the local let environment that would not be accessible at top level.

3.4.8 Example 3: The Callback Closure Problem

A frequently used programming model for library code is to allow an application to register a callback function for the library to call when some particular event occurs. It is often useful for the application to make several such registrations using the same callback function, for example if several similar library events can be handled using the same application code, but the need then arises to distinguish the callback function calls that are associated with one callback registration from those that are associated with different callback registrations.

In languages without the ability to create functions dynamically, this problem is usually solved by passing a user_data parameter on the registration call, and including the value of this parameter as one of the parameters on the callback function. Here is an example of declarations using this solution in C:

typedef void (event_handler_t) (int event_type,
  void *user_data);

void register_callback (int event_type,
  event_handler_t *handler,
  void *user_data);

In Scheme, closure can be used to achieve the same functionality without requiring the library code to store a user-data for each callback registration.

;; In the library:

(define (register-callback event-type handler-proc)
;; In the application:

(define (make-handler event-type user-data)
  (lambda ()
    ...
    <code referencing event-type and user-data>
    ...
))

(register-callback event-type
  (make-handler event-type ...))

As far as the library is concerned, handler-proc is a procedure with no arguments, and all the library has to do is call it when the appropriate event occurs. From the application’s point of view, though, the handler procedure has used closure to capture an environment that includes all the context that the handler code needs — event-type and user-data — to handle the event correctly.

3.4.9 Example 4: Object Orientation

Closure is the capture of an environment, containing persistent variable bindings, within the definition of a procedure or a set of related procedures. This is rather similar to the idea in some object oriented languages of encapsulating a set of related data variables inside an “object”, together with a set of “methods” that operate on the encapsulated data. The following example shows how closure can be used to emulate the ideas of objects, methods and encapsulation in Scheme.

(define (make-account)
  (let ((balance 0))
    (define (get-balance)
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (withdraw amount)
      (deposit (- amount)))

    (lambda args
      (apply
        (case (car args)
          ((get-balance) get-balance)
          ((deposit) deposit)
          ((withdraw) withdraw)
          (else (error "Invalid method!")))
        (cdr args))))

Each call to make-account creates and returns a new procedure, created by the expression in the example code that begins “(lambda args”).

(define my-account (make-account))
my-account
⇒
#<procedure args>

This procedure acts as an account object with methods get-balance, deposit and withdraw. To apply one of the methods to the account, you call the procedure with a symbol indicating the required method as the first parameter, followed by any other parameters that are required by that method.

\[(\text{my-account 'get-balance})\]
⇒
0

\[(\text{my-account 'withdraw 5})\]
⇒
-5

\[(\text{my-account 'deposit 396})\]
⇒
391

\[(\text{my-account 'get-balance})\]
⇒
391

Note how, in this example, both the current balance and the helper procedures get-balance, deposit and withdraw, used to implement the guts of the account object’s methods, are all stored in variable bindings within the private local environment captured by the lambda expression that creates the account object procedure.

3.5 Further Reading

- The website http://www.schemers.org/ is a good starting point for all things Scheme.
- Dorai Sitaram’s online Scheme tutorial, Teach Yourself Scheme in Fixnum Days, at http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html. Includes a nice explanation of continuations.
- The complete text of Structure and Interpretation of Computer Programs, the classic introduction to computer science and Scheme by Hal Abelson, Jerry Sussman and Julie Sussman, is now available online at http://mitpress.mit.edu/sicp/sicp.html. This site also provides teaching materials related to the book, and all the source code used in the book, in a form suitable for loading and running.
4 Programming in Scheme

Guile’s core language is Scheme, and a lot can be achieved simply by using Guile to write and run Scheme programs — as opposed to having to dive into C code. In this part of the manual, we explain how to use Guile in this mode, and describe the tools that Guile provides to help you with script writing, debugging, and packaging your programs for distribution.

For detailed reference information on the variables, functions, and so on that make up Guile’s application programming interface (API), see Chapter 6 [API Reference], page 99.

4.1 Guile’s Implementation of Scheme

Guile’s core language is Scheme, which is specified and described in the series of reports known as RnRS. RnRS is shorthand for the Revised* Report on the Algorithmic Language Scheme. Guile complies fully with R5RS (see Section “Introduction” in R5RS), and implements some aspects of R6RS.

Guile also has many extensions that go beyond these reports. Some of the areas where Guile extends R5RS are:

- Guile’s interactive documentation system
- Guile’s support for POSIX-compliant network programming
- GOOPS – Guile’s framework for object oriented programming.

4.2 Invoking Guile

Many features of Guile depend on and can be changed by information that the user provides either before or when Guile is started. Below is a description of what information to provide and how to provide it.

4.2.1 Command-line Options

Here we describe Guile’s command-line processing in detail. Guile processes its arguments from left to right, recognizing the switches described below. For examples, see Section 4.3.4 [Scripting Examples], page 44.

```
script arg...
-s script arg...
```

By default, Guile will read a file named on the command line as a script. Any command-line arguments arg... following script become the script’s arguments; the command-line function returns a list of strings of the form (script arg...).

It is possible to name a file using a leading hyphen, for example, -myfile.scm. In this case, the file name must be preceded by -s to tell Guile that a (script) file is being named.

Scripts are read and evaluated as Scheme source code just as the load function would. After loading script, Guile exits.

```
-c expr arg...
```

Evaluate expr as Scheme code, and then exit. Any command-line arguments arg... following expr become command-line arguments; the command-line func-
tion returns a list of strings of the form (guile arg...), where guile is the path of the Guile executable.

-- arg... Run interactively, prompting the user for expressions and evaluating them. Any command-line arguments arg... following the -- become command-line arguments for the interactive session; the command-line function returns a list of strings of the form (guile arg...), where guile is the path of the Guile executable.

-L directory
Add directory to the front of Guile’s module load path. The given directories are searched in the order given on the command line and before any directories in the GUILE_LOAD_PATH environment variable. Paths added here are not in effect during execution of the user’s .guile file.

-C directory
Like -L, but adjusts the load path for compiled files.

-x extension
Add extension to the front of Guile’s load extension list (see Section 6.18.7 [Load Paths], page 389). The specified extensions are tried in the order given on the command line, and before the default load extensions. Extensions added here are not in effect during execution of the user’s .guile file.

-l file
Load Scheme source code from file, and continue processing the command line.

-e function
Make function the entry point of the script. After loading the script file (with -s) or evaluating the expression (with -c), apply function to a list containing the program name and the command-line arguments—the list provided by the command-line function.

A -e switch can appear anywhere in the argument list, but Guile always invokes the function as the last action it performs. This is weird, but because of the way script invocation works under POSIX, the -s option must always come last in the list.

The function is most often a simple symbol that names a function that is defined in the script. It can also be of the form (@ module-name symbol), and in that case, the symbol is looked up in the module named module-name.

As a shorthand you can use the form (symbol ...), that is, a list of only symbols that doesn’t start with @. It is equivalent to (@ module-name main), where module-name is (symbol ...) form. See Section 6.20.2 [Using Guile Modules], page 407, and Section 4.3.4 [Scripting Examples], page 44.

-ds
Treat a final -s option as if it occurred at this point in the command line; load the script here.

This switch is necessary because, although the POSIX script invocation mechanism effectively requires the -s option to appear last, the programmer may well want to run the script before other actions requested on the command line. For examples, see Section 4.3.4 [Scripting Examples], page 44.
Read more command-line arguments, starting from the second line of the script file. See Section 4.3.2 [The Meta Switch], page 41.

--use-srfi=list

The option --use-srfi expects a comma-separated list of numbers, each representing a SRFI module to be loaded into the interpreter before evaluating a script file or starting the REPL. Additionally, the feature identifier for the loaded SRFIs is recognized by the procedure cond-expand when this option is used.

Here is an example that loads the modules SRFI-8 (‘receive’) and SRFI-13 (‘string library’) before the GUILE interpreter is started:

```
guile --use-srfi=8,13
```

--debug

Start with the debugging virtual machine (VM) engine. Using the debugging VM will enable support for VM hooks, which are needed for tracing, breakpoints, and accurate call counts when profiling. The debugging VM is slower than the regular VM, though, by about ten percent. See Section 6.26.4.1 [VM Hooks], page 481, for more information.

By default, the debugging VM engine is only used when entering an interactive session. When executing a script with -s or -c, the normal, faster VM is used by default.

--no-debug

Do not use the debugging VM engine, even when entering an interactive session. Note that, despite the name, Guile running with --no-debug does support the usual debugging facilities, such as printing a detailed backtrace upon error. The only difference with --debug is lack of support for VM hooks and the facilities that build upon it (see above).

-q

Do not load the initialization file, .guile. This option only has an effect when running interactively; running scripts does not load the .guile file. See Section 4.4.1 [Init File], page 48.

--listen=[p]

While this program runs, listen on a local port or a path for REPL clients. If p starts with a number, it is assumed to be a local port on which to listen. If it starts with a forward slash, it is assumed to be the file name of a UNIX domain socket on which to listen.

If p is not given, the default is local port 37146. If you look at it upside down, it almost spells “Guile”. If you have netcat installed, you should be able to nc localhost 37146 and get a Guile prompt. Alternately you can fire up Emacs and connect to the process; see Section 4.5 [Using Guile in Emacs], page 56, for more details.

**Note:** Opening a port allows anyone who can connect to that port to do anything Guile can do, as the user that the Guile process is running as. Do not use --listen on multi-user machines. Of course, if you do not pass --listen to Guile, no port will be opened.

Inter-protocol_exploitation), a scenario whereby an attacker can, via an HTML page, cause a web browser to send data to TCP servers listening on a loopback interface or private network. Nevertheless, you are advised to use UNIX domain sockets, as in --listen=/some/local/file, whenever possible.

That said, --listen is great for interactive debugging and development.

--auto-compile
Compile source files automatically (default behavior).

--fresh-auto-compile
Treat the auto-compilation cache as invalid, forcing recompilation.

--no-auto-compile
Disable automatic source file compilation.

--language=lang
For the remainder of the command line arguments, assume that files mentioned with -l and expressions passed with -c are written in lang. lang must be the name of one of the languages supported by the compiler (see Section 9.4.1 [Compiler Tower], page 836). When run interactively, set the REPL's language to lang (see Section 4.4 [Using Guile Interactively], page 47).

The default language is scheme; other interesting values include elisp (for Emacs Lisp), and ecmascript.

The example below shows the evaluation of expressions in Scheme, Emacs Lisp, and ECMAScript:

```
guile -c "(apply + '(1 2))"
guile --language=elisp -c "((funcall (symbol-function '+) 1 2) 3)"
guile --language=ecmascript -c 'function (x) { return x * x; })(2);'
```

To load a file written in Scheme and one written in Emacs Lisp, and then start a Scheme REPL, type:

```
guile -l foo.scm --language=elisp -l foo.el --language=scheme
```

-h, --help
Display help on invoking Guile, and then exit.

-v, --version
Display the current version of Guile, and then exit.

4.2.2 Environment Variables

The environment is a feature of the operating system; it consists of a collection of variables with names and values. Each variable is called an environment variable (or, sometimes, a "shell variable"); environment variable names are case-sensitive, and it is conventional to use upper-case letters only. The values are all text strings, even those that are written as numerals. (Note that here we are referring to names and values that are defined in the operating system shell from which Guile is invoked. This is not the same as a Scheme environment that is defined within a running instance of Guile. For a description of Scheme environments, see Section 3.4.1 [About Environments], page 26.)
How to set environment variables before starting Guile depends on the operating system and, especially, the shell that you are using. For example, here is how to tell Guile to provide detailed warning messages about deprecated features by setting `GUILE_WARN_DEPRECATED` using Bash:

```
$ export GUILE_WARN_DEPRECATED="detailed"
$ guile
```

Or, detailed warnings can be turned on for a single invocation using:

```
$ env GUILE_WARN_DEPRECATED="detailed" guile
```

If you wish to retrieve or change the value of the shell environment variables that affect the run-time behavior of Guile from within a running instance of Guile, see Section 7.2.6 [Runtime Environment], page 512.

Here are the environment variables that affect the run-time behavior of Guile:

**GUILE_AUTO_COMPILE**
This is a flag that can be used to tell Guile whether or not to compile Scheme source files automatically. Starting with Guile 2.0, Scheme source files will be compiled automatically, by default.

If a compiled (.go) file corresponding to a .scm file is not found or is not newer than the .scm file, the .scm file will be compiled on the fly, and the resulting .go file stored away. An advisory note will be printed on the console.

Compiled files will be stored in the directory `$XDG_CACHE_HOME/guile/ccache`, where `XDG_CACHE_HOME` defaults to the directory `$HOME/.cache`. This directory will be created if it does not already exist.

Note that this mechanism depends on the timestamp of the .go file being newer than that of the .scm file; if the .scm or .go files are moved after installation, care should be taken to preserve their original timestamps.

Set `GUILE_AUTO_COMPILE` to zero (0), to prevent Scheme files from being compiled automatically. Set this variable to “fresh” to tell Guile to compile Scheme files whether they are newer than the compiled files or not.

See Section 6.18.5 [Compilation], page 386.

**GUILE_HISTORY**
This variable names the file that holds the Guile REPL command history. You can specify a different history file by setting this environment variable. By default, the history file is `$HOME/.guile_history`.

**GUILE_INSTALL_LOCALE**
This is a flag that can be used to tell Guile whether or not to install the current locale at startup, via a call to `(setlocale LC_ALL "")`\(^1\). See Section 7.2.13 [Locales], page 542, for more information on locales.

You may explicitly indicate that you do not want to install the locale by setting `GUILE_INSTALL_LOCALE` to 0, or explicitly enable it by setting the variable to 1.

---

\(^1\) The `GUILE_INSTALL_LOCALE` environment variable was ignored in Guile versions prior to 2.0.9.
Usually, installing the current locale is the right thing to do. It allows Guile to correctly parse and print strings with non-ASCII characters. Therefore, this option is on by default.

**GUILE STACK SIZE**
Guile currently has a limited stack size for Scheme computations. Attempting to call too many nested functions will signal an error. This is good to detect infinite recursion, but sometimes the limit is reached for normal computations. This environment variable, if set to a positive integer, specifies the number of Scheme value slots to allocate for the stack.

In the future we will implement stacks that can grow and shrink, but for now this hack will have to do.

**GUILE_LOAD_COMPILED_PATH**
This variable may be used to augment the path that is searched for compiled Scheme files (.go files) when loading. Its value should be a colon-separated list of directories. If it contains the special path component ... (ellipsis), then the default path is put in place of the ellipsis, otherwise the default path is placed at the end. The result is stored in `%load-compiled-path` (see Section 6.18.7 [Load Paths], page 389).

Here is an example using the Bash shell that adds the current directory, .., and the relative directory ../my-library to `%load-compiled-path`:

```
$ export GUILE_LOAD_COMPILED_PATH="..:/my-library"
$ guile -c '(display `%load-compiled-path`) (newline)'
(../my-library /usr/local/lib/guile/2.2/ccache)
```

**GUILE_LOAD_PATH**
This variable may be used to augment the path that is searched for Scheme files when loading. Its value should be a colon-separated list of directories. If it contains the special path component ... (ellipsis), then the default path is put in place of the ellipsis, otherwise the default path is placed at the end. The result is stored in `%load-path` (see Section 6.18.7 [Load Paths], page 389).

Here is an example using the Bash shell that prepends the current directory to `%load-path`, and adds the relative directory ../srfi to the end:

```
$ env GUILE_LOAD_PATH=".......:/srfi" 
guile -c '(display `%load-path`) (newline)'
(./ /usr/local/share/guile/2.2 
 /usr/local/share/guile/site/2.2 
 /usr/local/share/guile/site 
 /usr/local/share/guile 
 ../srfi)
```

(Note: The line breaks, above, are for documentation purposes only, and not required in the actual example.)

**GUILE_WARN_DEPRECATED**
As Guile evolves, some features will be eliminated or replaced by newer features. To help users migrate their code as this evolution occurs, Guile will issue warning messages about code that uses features that have been marked for eventual
Chapter 4: Programming in Scheme

elimination. The environment variable `HOME` can be set to “no” to tell Guile not to display these warning messages, or set to “detailed” to tell Guile to display more lengthy messages describing the warning. See Section 6.2 [Deprecation], page 100.

HOME

Guile uses the environment variable `HOME`, the name of your home directory, to locate various files, such as `.guile` or `.guile_history`.

4.3 Guile Scripting

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

4.3.1 The Top of a Script File

The first line of a Guile script must tell the operating system to use Guile to evaluate the script, and then tell Guile how to go about doing that. Here is the simplest case:

- The first two characters of the file must be `#!/`.
  
  The operating system interprets this to mean that the rest of the line is the name of an executable that can interpret the script. Guile, however, interprets these characters as the beginning of a multi-line comment, terminated by the characters `!#` on a line by themselves. (This is an extension to the syntax described in R5RS, added to support shell scripts.)

- Immediately after those two characters must come the full pathname to the Guile interpreter. On most systems, this would be `/usr/local/bin/guile`.

- Then must come a space, followed by a command-line argument to pass to Guile; this should be `--s`. This switch tells Guile to run a script, instead of soliciting the user for input from the terminal. There are more elaborate things one can do here; see Section 4.3.2 [The Meta Switch], page 41.

- Follow this with a newline.

- The second line of the script should contain only the characters `!#` — just like the top of the file, but reversed. The operating system never reads this far, but Guile treats this as the end of the comment begun on the first line by the `#!/` characters.

- If this source code file is not ASCII or ISO-8859-1 encoded, a coding declaration such as `coding: utf-8` should appear in a comment somewhere in the first five lines of the file: see Section 6.18.8 [Character Encoding of Source Files], page 392.

- The rest of the file should be a Scheme program.

  Guile reads the program, evaluating expressions in the order that they appear. Upon reaching the end of the file, Guile exits.

4.3.2 The Meta Switch

Guile’s command-line switches allow the programmer to describe reasonably complicated actions in scripts. Unfortunately, the POSIX script invocation mechanism only allows one argument to appear on the `#!/` line after the path to the Guile executable, and imposes arbitrary limits on that argument’s length. Suppose you wrote a script starting like this:

`#!/usr/local/bin/guile --e main --s`
#!
(define (main args)
  (map (lambda (arg) (display arg) (display " "))
    (cdr args))
  (newline))

The intended meaning is clear: load the file, and then call main on the command-line arguments. However, the system will treat everything after the Guile path as a single argument — the string "-e main -s" — which is not what we want.

As a workaround, the meta switch \ allows the Guile programmer to specify an arbitrary number of options without patching the kernel. If the first argument to Guile is \, Guile will open the script file whose name follows the \, parse arguments starting from the file's second line (according to rules described below), and substitute them for the \ switch.

Working in concert with the meta switch, Guile treats the characters ‘#!’ as the beginning of a comment which extends through the next line containing only the characters ‘!#’. This sort of comment may appear anywhere in a Guile program, but it is most useful at the top of a file, meshing magically with the POSIX script invocation mechanism.

Thus, consider a script named /u/jimb/ekko which starts like this:

```
#!/usr/local/bin/guile \ 
-e main -s
#!
(define (main args)
  (map (lambda (arg) (display arg) (display " "))
    (cdr args))
  (newline))
```

Suppose a user invokes this script as follows:

```
$ /u/jimb/ekko a b c
```

Here's what happens:

- the operating system recognizes the ‘#!’ token at the top of the file, and rewrites the command line to:

```
/usr/local/bin/guile \ /u/jimb/ekko a b c
```

This is the usual behavior, prescribed by POSIX.

- When Guile sees the first two arguments, \ /u/jimb/ekko, it opens /u/jimb/ekko, parses the three arguments -e, main, and -s from it, and substitutes them for the \ switch. Thus, Guile's command line now reads:

```
/usr/local/bin/guile -e main -s /u/jimb/ekko a b c
```

- Guile then processes these switches: it loads /u/jimb/ekko as a file of Scheme code (treating the first three lines as a comment), and then performs the application (main "/u/jimb/ekko" "a" "b" "c").

When Guile sees the meta switch \, it parses command-line argument from the script file according to the following rules:

- Each space character terminates an argument. This means that two spaces in a row introduce an argument " ".
The tab character is not permitted (unless you quote it with the backslash character, as described below), to avoid confusion.

The newline character terminates the sequence of arguments, and will also terminate a final non-empty argument. (However, a newline following a space will not introduce a final empty-string argument; it only terminates the argument list.)

The backslash character is the escape character. It escapes backslash, space, tab, and newline. The ANSI C escape sequences like \n and \t are also supported. These produce argument constituents; the two-character combination \n doesn’t act like a terminating newline. The escape sequence \NNN for exactly three octal digits reads as the character whose ASCII code is NNN. As above, characters produced this way are argument constituents. Backslash followed by other characters is not allowed.

4.3.3 Command Line Handling

The ability to accept and handle command line arguments is very important when writing Guile scripts to solve particular problems, such as extracting information from text files or interfacing with existing command line applications. This chapter describes how Guile makes command line arguments available to a Guile script, and the utilities that Guile provides to help with the processing of command line arguments.

When a Guile script is invoked, Guile makes the command line arguments accessible via the procedure command-line, which returns the arguments as a list of strings.

For example, if the script

```scheme
#!/usr/local/bin/guile -s

(write (command-line))

(newline)
```

is saved in a file cmdline-test.scm and invoked using the command line

```
./cmdline-test.scm bar.txt -o foo -frumple grob
```

the output is

```scheme
("./cmdline-test.scm" "bar.txt" "-o" "foo" "-frumple" "grob")
```

If the script invocation includes a -e option, specifying a procedure to call after loading the script, Guile will call that procedure with (command-line) as its argument. So a script that uses -e doesn’t need to refer explicitly to command-line in its code. For example, the script above would have identical behaviour if it was written instead like this:

```scheme
#!/usr/local/bin/guile \
-e main -s

(define (main args)
  (write args)
  (newline))
```

(Note the use of the meta switch \ so that the script invocation can include more than one Guile option: See Section 4.3.2 [The Meta Switch], page 41.)

These scripts use the #! POSIX convention so that they can be executed using their own file names directly, as in the example command line ./cmdline-test.scm bar.txt -o foo -frumple grob. But they can also be executed by typing out the implied Guile command line in full, as in:

```
$ guile -s ./cmdline-test.scm bar.txt -o foo -frumple grob
```
or

$ guile -e main -s ./cmdline-test2.scm bar.txt -o foo -frumple grob

Even when a script is invoked using this longer form, the arguments that the script receives are the same as if it had been invoked using the short form. Guile ensures that the (command-line) or -e arguments are independent of how the script is invoked, by stripping off the arguments that Guile itself processes.

A script is free to parse and handle its command line arguments in any way that it chooses. Where the set of possible options and arguments is complex, however, it can get tricky to extract all the options, check the validity of given arguments, and so on. This task can be greatly simplified by taking advantage of the module (ice-9 getopt-long), which is distributed with Guile, See Section 7.4 [getopt-long], page 571.

### 4.3.4 Scripting Examples

To start with, here are some examples of invoking Guile directly:

**guile -- a b c**

Run Guile interactively; (command-line) will return

"/usr/local/bin/guile" "a" "b" "c"

**guile -s /u/jimb/ex2 a b c**

Load the file /u/jimb/ex2; (command-line) will return

"/u/jimb/ex2" "a" "b" "c"

**guile -c '(write %load-path) (newline)'**

Write the value of the variable %load-path, print a newline, and exit.

**guile -e main -s /u/jimb/ex4 foo**

Load the file /u/jimb/ex4, and then call the function main, passing it the list

"/u/jimb/ex4" "foo"

**guile -e '(ex4)' -s /u/jimb/ex4.scm foo**

Load the file /u/jimb/ex4.scm, and then call the function main from the module '(ex4)', passing it the list

"/u/jimb/ex4" "foo"

**guile -l first -ds -l last -s script**

Load the files first, script, and last, in that order. The -ds switch says when to process the -s switch. For a more motivated example, see the scripts below.

Here is a very simple Guile script:

```guile
#!/usr/local/bin/guile -s
#
(display "Hello, world!")
(newline)
```

The first line marks the file as a Guile script. When the user invokes it, the system runs /usr/local/bin/guile to interpret the script, passing -s, the script’s filename, and any arguments given to the script as command-line arguments. When Guile sees -s script, it loads script. Thus, running this program produces the output:

Hello, world!
Here is a script which prints the factorial of its argument:

```scheme
#!/usr/local/bin/guile -s
!(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(display (fact (string->number (cadr (command-line)))))
(newline)
```

In action:

```
$ ./fact 5
120
$
```

However, suppose we want to use the definition of `fact` in this file from another script. We can’t simply load the script file, and then use `fact`’s definition, because the script will try to compute and display a factorial when we load it. To avoid this problem, we might write the script this way:

```scheme
#!/usr/local/bin/guile \
-e main -s
!(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(define (main args)
  (display (fact (string->number (cadr args))))
  (newline))
```

This version packages the actions the script should perform in a function, `main`. This allows us to load the file purely for its definitions, without any extraneous computation taking place. Then we used the meta switch `\` and the entry point switch `-e` to tell Guile to call `main` after loading the script.

```
$ ./fact 50
30414093201713378043612608166064768844377641568960512000000000000000000
```

Suppose that we now want to write a script which computes the `choose` function: given a set of `m` distinct objects, `(choose n m)` is the number of distinct subsets containing `n` objects each. It’s easy to write `choose` given `fact`, so we might write the script this way:

```scheme
#!/usr/local/bin/guile \
-l fact -e main -s
!(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
         (m (string->number (cadr args))))
```
The command-line arguments here tell Guile to first load the file fact, and then run the
script, with main as the entry point. In other words, the choose script can use definitions
made in the fact script. Here are some sample runs:

$ ./choose 0 4
1
$ ./choose 1 4
4
$ ./choose 2 4
6
$ ./choose 3 4
4
$ ./choose 4 4
1
$ ./choose 50 100
100891344545564193334812497256

To call a specific procedure from a given module, we can use the special form (@ (module)
procedure):

#!/usr/local/bin/guile \
-l fact -e (@ (fac) main) -s
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))

We can use @@ to invoke non-exported procedures. For exported procedures, we can
simplify this call with the shorthand (module):

#!/usr/local/bin/guile \
-l fact -e (fac) -s
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
(let ((n (string->number (cadr args)))
    (m (string->number (caddr args))))
  (display (choose n m))
  (newline))

For maximum portability, we can instead use the shell to execute guile with specified command line arguments. Here we need to take care to quote the command arguments correctly:

```sh
#!/usr/bin/env sh
exec guile -l fact -e '@ (fac) main)' -s "$0" "$@
```

```scheme
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline))
)
```

Finally, seasoned scripters are probably missing a mention of subprocesses. In Bash, for example, most shell scripts run other programs like sed or the like to do the actual work.

In Guile it’s often possible get everything done within Guile itself, so do give that a try first. But if you just need to run a program and wait for it to finish, use system*. If you need to run a sub-program and capture its output, or give it input, use open-pipe. See Section 7.2.7 [Processes], page 514, and See Section 7.2.10 [Pipes], page 524, for more information.

### 4.4 Using Guile Interactively

When you start up Guile by typing just guile, without a -c argument or the name of a script to execute, you get an interactive interpreter where you can enter Scheme expressions, and Guile will evaluate them and print the results for you. Here are some simple examples.

```scheme
scheme@(guile-user)> (+ 3 4 5)
$1 = 12
scheme@(guile-user)> (display "Hello world!\n")
Hello world!
scheme@(guile-user)> (values 'a 'b)
$2 = a
$3 = b
```

This mode of use is called a REPL, which is short for “Read-Eval-Print Loop”, because the Guile interpreter first reads the expression that you have typed, then evaluates it, and then prints the result.

The prompt shows you what language and module you are in. In this case, the current language is scheme, and the current module is (guile-user).
4.4.1 The Init File, ~/.guile

When run interactively, Guile will load a local initialization file from ~/.guile. This file should contain Scheme expressions for evaluation.

This facility lets the user customize their interactive Guile environment, pulling in extra modules or parameterizing the REPL implementation.

To run Guile without loading the init file, use the -q command-line option.

4.4.2 Readline

To make it easier for you to repeat and vary previously entered expressions, or to edit the expression that you’re typing in, Guile can use the GNU Readline library. This is not enabled by default because of licensing reasons, but all you need to activate Readline is the following pair of lines.

scheme@(guile-user)> (use-modules (ice-9 readline))
scheme@(guile-user)> (activate-readline)

It’s a good idea to put these two lines (without the scheme@(guile-user)> prompts) in your .guile file. See Section 4.4.1 [Init File], page 48, for more on .guile.

4.4.3 Value History

Just as Readline helps you to reuse a previous input line, value history allows you to use the result of a previous evaluation in a new expression. When value history is enabled, each evaluation result is automatically assigned to the next in the sequence of variables $1$, $2$, ... You can then use these variables in subsequent expressions.

scheme@(guile-user)> (iota 10)
$1 = (0 1 2 3 4 5 6 7 8 9)
scheme@(guile-user)> (apply * (cdr $1))
$2 = 362880
scheme@(guile-user)> (sqrt $2)
$3 = 602.3952191045344
scheme@(guile-user)> (cons $2 $1)
$4 = (362880 0 1 2 3 4 5 6 7 8 9)

Value history is enabled by default, because Guile’s REPL imports the (ice-9 history) module. Value history may be turned off or on within the repl, using the options interface:

scheme@(guile-user)> ,option value-history #f
scheme@(guile-user)> 'foo
foo
scheme@(guile-user)> ,option value-history #t
scheme@(guile-user)> 'bar
$5 = bar

Note that previously recorded values are still accessible, even if value history is off. In rare cases, these references to past computations can cause Guile to use too much memory. One may clear these values, possibly enabling garbage collection, via the clear-value-history! procedure, described below.
The programmatic interface to value history is in a module:

(use-modules (ice-9 history))

value-history-enabled? [Scheme Procedure]
Return true if value history is enabled, or false otherwise.

enable-value-history! [Scheme Procedure]
Turn on value history, if it was off.

disable-value-history! [Scheme Procedure]
Turn off value history, if it was on.

clear-value-history! [Scheme Procedure]
Clear the value history. If the stored values are not captured by some other data
structure or closure, they may then be reclaimed by the garbage collector.

4.4.4 REPL Commands
The REPL exists to read expressions, evaluate them, and then print their results. But
sometimes one wants to tell the REPL to evaluate an expression in a different way, or to
do something else altogether. A user can affect the way the REPL works with a REPL
command.

The previous section had an example of a command, in the form of ,option.

scheme@(guile-user)> ,option value-history #t
Commands are distinguished from expressions by their initial comma (', '). Since a comma
cannot begin an expression in most languages, it is an effective indicator to the REPL that
the following text forms a command, not an expression.

REPL commands are convenient because they are always there. Even if the current
module doesn’t have a binding for pretty-print, one can always ,pretty-print.

The following sections document the various commands, grouped together by function-
ality. Many of the commands have abbreviations; see the online help (,help) for more
information.

4.4.4.1 Help Commands
When Guile starts interactively, it notifies the user that help can be had by typing ‘,help’. Indeed, help is a command, and a particularly useful one, as it allows the user to discover
the rest of the commands.

help [all | group | [-c] command] [REPL Command]
Show help.

With one argument, tries to look up the argument as a group name, giving help on
that group if successful. Otherwise tries to look up the argument as a command,
giving help on the command.

If there is a command whose name is also a group name, use the ‘-c command’ form
to give help on the command instead of the group.

Without any argument, a list of help commands and command groups are displayed.
show [topic]  [REPL Command]
   Gives information about Guile.
   With one argument, tries to show a particular piece of information; currently sup-
   ported topics are ‘warranty’ (or ‘w’), ‘copying’ (or ‘c’), and ‘version’ (or ‘v’).
   Without any argument, a list of topics is displayed.

apropos regexp  [REPL Command]
   Find bindings/modules/packages.

describe obj  [REPL Command]
   Show description/documentation.

4.4.4.2 Module Commands

module [module]  [REPL Command]
   Change modules / Show current module.

import module . . .  [REPL Command]
   Import modules / List those imported.

load file  [REPL Command]
   Load a file in the current module.

reload [module]  [REPL Command]
   Reload the given module, or the current module if none was given.

binding  [REPL Command]
   List current bindings.

in module expression  [REPL Command]

in module command arg . . .  [REPL Command]
   Evaluate an expression, or alternatively, execute another meta-command in the con-
   text of a module. For example, ‘,in (foo bar) ,binding’ will show the bindings in
   the module (foo bar).

4.4.4.3 Language Commands

language language  [REPL Command]
   Change languages.

4.4.4.4 Compile Commands

compile exp  [REPL Command]
   Generate compiled code.

compile-file file  [REPL Command]
   Compile a file.

expand exp  [REPL Command]
   Expand any macros in a form.
**optimize** `exp`  
Run the optimizer on a piece of code and print the result.

**disassemble** `exp`  
Disassemble a compiled procedure.

**disassemble-file** `file`  
Disassemble a file.

### 4.4.4.5 Profile Commands

**time** `exp`  
Time execution.

**profile** `exp` [#:hz `hz=100`][#:count-calls? `count-calls?=#f`][#:display-style `display-style=list`]

Profile execution of an expression. This command compiled `exp` and then runs it within the `statprof` profiler, passing all keyword options to the `statprof` procedure. For more on `statprof` and on the the options available to this command, See Section 7.19 [Statprof], page 729.

**trace** `exp` [#:width `w`][#:max-indent `i`]

Trace execution.

By default, the trace will limit its width to the width of your terminal, or `width` if specified. Nested procedure invocations will be printed farther to the right, though if the width of the indentation passes the `max-indent`, the indentation is abbreviated.

### 4.4.4.6 Debug Commands

These debugging commands are only available within a recursive REPL; they do not work at the top level.

**backtrace** `[count]`[#:width `w`][#:full? `f`]

Print a backtrace.

Print a backtrace of all stack frames, or innermost `count` frames. If `count` is negative, the last `count` frames will be shown.

**up** `[count]`

Select a calling stack frame.

Select and print stack frames that called this one. An argument says how many frames up to go.

**down** `[count]`

Select a called stack frame.

Select and print stack frames called by this one. An argument says how many frames down to go.

**frame** `[idx]`

Show a frame.

Show the selected frame. With an argument, select a frame by index, then show it.
locals  [REPL Command]
Show local variables.
Show locally-bound variables in the selected frame.

error-message  [REPL Command]
error  [REPL Command]
Show error message.
Display the message associated with the error that started the current debugging REPL.

registers  [REPL Command]
Show the VM registers associated with the current frame.
See Section 9.3.3 [Stack Layout], page 814, for more information on VM stack frames.

width [cols]  [REPL Command]
Sets the number of display columns in the output of ,backtrace and ,locals to cols. If cols is not given, the width of the terminal is used.

The next 3 commands work at any REPL.

break proc  [REPL Command]
Set a breakpoint at proc.

break-at-source file line  [REPL Command]
Set a breakpoint at the given source location.

tracepoint proc  [REPL Command]
Set a tracepoint on the given procedure. This will cause all calls to the procedure to print out a tracing message. See Section 6.26.4.4 [Tracing Traps], page 485, for more information.

The rest of the commands in this subsection all apply only when the stack is continuable — in other words when it makes sense for the program that the stack comes from to continue running. Usually this means that the program stopped because of a trap or a breakpoint.

step  [REPL Command]
Tell the debugged program to step to the next source location.

next  [REPL Command]
Tell the debugged program to step to the next source location in the same frame. (See Section 6.26.4 [Traps], page 480, for the details of how this works.)

finish  [REPL Command]
Tell the program being debugged to continue running until the completion of the current stack frame, and at that time to print the result and reenter the REPL.

4.4.4.7 Inspect Commands

inspect exp  [REPL Command]
Inspect the result(s) of evaluating exp.

pretty-print exp  [REPL Command]
Pretty-print the result(s) of evaluating exp.
4.4.4.8 System Commands

gc  [REPL Command]
Garbage collection.

statistics  [REPL Command]
Display statistics.

option [name] [exp]  [REPL Command]
With no arguments, lists all options. With one argument, shows the current value
of the name option. With two arguments, sets the name option to the result of
evaluating the Scheme expression exp.

quit  [REPL Command]
Quit this session.

Current REPL options include:

compile-options
The options used when compiling expressions entered at the REPL. See
Section 6.18.5 [Compilation], page 386, for more on compilation options.

interp
Whether to interpret or compile expressions given at the REPL, if such a choice
is available. Off by default (indicating compilation).

prompt
A customized REPL prompt. #f by default, indicating the default prompt.

print
A procedure of two arguments used to print the result of evaluating each ex-
pression. The arguments are the current REPL and the value to print. By
default, #f, to use the default procedure.

value-history
Whether value history is on or not. See Section 4.4.3 [Value History], page 48.

on-error
What to do when an error happens. By default, debug, meaning to enter
the debugger. Other values include backtrace, to show a backtrace without
entering the debugger, or report, to simply show a short error printout.

Default values for REPL options may be set using repl-default-option-set! from
(system repl common):

repl-default-option-set! key value  [Scheme Procedure]
Set the default value of a REPL option. This function is particularly useful in a user’s
init file. See Section 4.4.1 [Init File], page 48.

4.4.5 Error Handling

When code being evaluated from the REPL hits an error, Guile enters a new prompt,
allowing you to inspect the context of the error.

scheme@(guile-user)> (map string-append '("a" "b") '("c" #\d))
ERROR: In procedure string-append:
ERROR: Wrong type (expecting string): #\d
Entering a new prompt. Type ',bt' for a backtrace or ',q' to continue.
scheme@(guile-user) [1]>
The new prompt runs inside the old one, in the dynamic context of the error. It is a recursive REPL, augmented with a reified representation of the stack, ready for debugging.

,backtrace (abbreviated ,bt) displays the Scheme call stack at the point where the error occurred:

scheme@(guile-user) [1]> ,bt
1 (map #<procedure string-append _> ("a" "b") ("c" #\d))
0 (string-append "b" #\d)

In the above example, the backtrace doesn’t have much source information, as map and string-append are both primitives. But in the general case, the space on the left of the backtrace indicates the line and column in which a given procedure calls another.

You can exit a recursive REPL in the same way that you exit any REPL: via ‘(quit)’, ‘,quit’ (abbreviated ‘,q’), or C-d, among other options.

4.4.6 Interactive Debugging
A recursive debugging REPL exposes a number of other meta-commands that inspect the state of the computation at the time of the error. These commands allow you to
- display the Scheme call stack at the point where the error occurred;
- move up and down the call stack, to see in detail the expression being evaluated, or the procedure being applied, in each frame; and
- examine the values of variables and expressions in the context of each frame.

See Section 4.4.4.6 [Debug Commands], page 51, for documentation of the individual commands. This section aims to give more of a walkthrough of a typical debugging session.

First, we’re going to need a good error. Let’s try to macroexpand the expression (unquote foo), outside of a quasiquote form, and see how the macroexpander reports this error.

scheme@(guile-user)> (macroexpand '(unquote foo))
ERROR: In procedure macroexpand:
ERROR: unquote: expression not valid outside of quasiquote in (unquote foo)
Entering a new prompt. Type ‘,bt’ for a backtrace or ‘,q’ to continue.
scheme@(guile-user) [1]>

The backtrace command, which can also be invoked as bt, displays the call stack (aka backtrace) at the point where the debugger was entered:

scheme@(guile-user) [1]> ,bt
In ice-9/psyntax.scm:
1130:21 3 (chi-top (unquote foo) () ((top)) e (eval) (hygiene #))
1071:30 2 (syntax-type (unquote foo) () ((top)) #f #f (# #) #f)
1368:28 1 (chi-macro #<procedure de9360 at ice-9/psyntax.scm...> ...) In unknown file:
0 (scm-error syntax-error macroexpand "^a: ^a in ^a # #f")

A call stack consists of a sequence of stack frames, with each frame describing one procedure which is waiting to do something with the values returned by another. Here we see that there are four frames on the stack.
Note that `macroexpand` is not on the stack – it must have made a tail call to `chi-top`, as indeed we would find if we searched `ice-9/psyntax.scm` for its definition.

When you enter the debugger, the innermost frame is selected, which means that the commands for getting information about the “current” frame, or for evaluating expressions in the context of the current frame, will do so by default with respect to the innermost frame. To select a different frame, so that these operations will apply to it instead, use the `up`, `down` and `frame` commands like this:

```scheme
scheme@(guile-user) [1]> ,up
In ice-9/psyntax.scm:
  1368:28  1 (chi-macro #<procedure de9360 at ice-9/psyntax.scm...> ...)
scheme@(guile-user) [1]> ,frame 3
In ice-9/psyntax.scm:
  1130:21  3 (chi-top (unquote foo) () ((top)) e (eval) (hygiene #))
scheme@(guile-user) [1]> ,down
In ice-9/psyntax.scm:
  1071:30  2 (syntax-type (unquote foo) () ((top)) #f #f (# #) #f)
```

Perhaps we're interested in what's going on in frame 2, so we take a look at its local variables:

```scheme
scheme@(guile-user) [1]> ,locals
Local variables:
$1 = e = (unquote foo)
$2 = r = ()
$3 = w = ((top))
$4 = s = #f
$5 = rib = #f
$6 = mod = (hygiene guile-user)
$7 = for-car? = #f
$8 = first = unquote
$9 = ftype = macro
$10 = fval = #<procedure de9360 at ice-9/psyntax.scm:2817:2 (x)>  
$11 = fe = unquote
$12 = fw = ((top))
$13 = fs = #f
$14 = fmod = (hygiene guile-user)
```

All of the values are accessible by their value-history names ($n$):

```scheme
scheme@(guile-user) [1]> $10
$15 = #<procedure de9360 at ice-9/psyntax.scm:2817:2 (x)>
```

We can even invoke the procedure at the REPL directly:

```scheme
scheme@(guile-user) [1]> ($10 'not-going-to-work)
ERROR: In procedure macroexpand:
  ERROR: source expression failed to match any pattern in not-going-to-work
Entering a new prompt. Type ",bt" for a backtrace or ",q" to continue.
```

Well at this point we've caused an error within an error. Let's just quit back to the top level:

```scheme
scheme@(guile-user) [2]> ,q
```
Finally, as a word to the wise: hackers close their REPL prompts with \texttt{C-d}.

### 4.5 Using Guile in Emacs

Any text editor can edit Scheme, but some are better than others. Emacs is the best, of course, and not just because it is a fine text editor. Emacs has good support for Scheme out of the box, with sensible indentation rules, parenthesis-matching, syntax highlighting, and even a set of keybindings for structural editing, allowing navigation, cut-and-paste, and transposition operations that work on balanced S-expressions.

As good as it is, though, two things will vastly improve your experience with Emacs and Guile.

The first is Taylor Campbell’s Paredit (\texttt{http://www.emacswiki.org/emacs/ParEdit}). You should not code in any dialect of Lisp without Paredit. (They say that unopinionated writing is boring—hence this tone—but it’s the truth, regardless.) Paredit is the bee’s knees.

The second is José Antonio Ortega Ruiz’s Geiser (\texttt{http://www.nongnu.org/geiser/}). Geiser complements Emacs’ \texttt{scheme-mode} with tight integration to running Guile processes via a \texttt{comint-mode} REPL buffer.

Of course there are keybindings to switch to the REPL, and a good REPL environment, but Geiser goes beyond that, providing:

- Form evaluation in the context of the current file’s module.
- Macro expansion.
- File/module loading and/or compilation.
- Namespace-aware identifier completion (including local bindings, names visible in the current module, and module names).
- Autodoc: the echo area shows information about the signature of the procedure/macro around point automatically.
- Jump to definition of identifier at point.
- Access to documentation (including docstrings when the implementation provides it).
- Listings of identifiers exported by a given module.
- Listings of callers/callees of procedures.
- Rudimentary support for debugging and error navigation.
- Support for multiple, simultaneous REPLs.

See Geiser’s web page at \texttt{http://www.nongnu.org/geiser/}, for more information.

### 4.6 Using Guile Tools

Guile also comes with a growing number of command-line utilities: a compiler, a disassembler, some module inspectors, and in the future, a system to install Guile packages from the internet. These tools may be invoked using the \texttt{guild} program.

```
$ guild compile -o foo.go foo.scm
```
This program used to be called guile-tools up to Guile version 2.0.1, and for backward compatibility it still may be called as such. However we changed the name to guild, not only because it is pleasantly shorter and easier to read, but also because this tool will serve to bind Guile wizards together, by allowing hackers to share code with each other using a CPAN-like system.

See Section 6.18.5 [Compilation], page 386, for more on guild compile.

A complete list of guild scripts can be had by invoking guild list, or simply guild.

4.7 Installing Site Packages

At some point, you will probably want to share your code with other people. To do so effectively, it is important to follow a set of common conventions, to make it easy for the user to install and use your package.

The first thing to do is to install your Scheme files where Guile can find them. When Guile goes to find a Scheme file, it will search a load path to find the file: first in Guile’s own path, then in paths for site packages. A site package is any Scheme code that is installed and not part of Guile itself. See Section 6.18.7 [Load Paths], page 389, for more on load paths.

There are several site paths, for historical reasons, but the one that should generally be used can be obtained by invoking the %site-dir procedure. See Section 6.23.1 [Build Config], page 450. If Guile 2.2 is installed on your system in /usr/, then (%site-dir) will be /usr/share/guile/site/2.2. Scheme files should be installed there.

If you do not install compiled .go files, Guile will compile your modules and programs when they are first used, and cache them in the user’s home directory. See Section 6.18.5 [Compilation], page 386, for more on auto-compilation. However, it is better to compile the files before they are installed, and to just copy the files to a place that Guile can find them.

As with Scheme files, Guile searches a path to find compiled .go files, the %load-compiled-path. By default, this path has two entries: a path for Guile’s files, and a path for site packages. You should install your .go files into the latter directory, whose value is returned by invoking the %site-ccache-dir procedure. As in the previous example, if Guile 2.2 is installed on your system in /usr/, then (%site-ccache-dir) site packages will be /usr/lib/guile/2.2/site-ccache.

Note that a .go file will only be loaded in preference to a .scm file if it is newer. For that reason, you should install your Scheme files first, and your compiled files second. See Section 6.18.7 [Load Paths], page 389, for more on the loading process.

Finally, although this section is only about Scheme, sometimes you need to install C extensions too. Shared libraries should be installed in the extensions dir. This value can be had from the build config (see Section 6.23.1 [Build Config], page 450). Again, if Guile 2.2 is installed on your system in /usr/, then the extensions dir will be /usr/lib/guile/2.2/ extensions.
5 Programming in C

This part of the manual explains the general concepts that you need to understand when interfacing to Guile from C. You will learn about how the latent typing of Scheme is embedded into the static typing of C, how the garbage collection of Guile is made available to C code, and how continuations influence the control flow in a C program.

This knowledge should make it straightforward to add new functions to Guile that can be called from Scheme. Adding new data types is also possible and is done by defining foreign objects.

The Section 5.7 [Programming Overview], page 83, section of this part contains general musings and guidelines about programming with Guile. It explores different ways to design a program around Guile, or how to embed Guile into existing programs.

For a pedagogical yet detailed explanation of how the data representation of Guile is implemented, See Section 9.2 [Data Representation], page 803. You don’t need to know the details given there to use Guile from C, but they are useful when you want to modify Guile itself or when you are just curious about how it is all done.

For detailed reference information on the variables, functions etc. that make up Guile’s application programming interface (API), See Chapter 6 [API Reference], page 99.

5.1 Parallel Installations

Guile provides strong API and ABI stability guarantees during stable series, so that if a user writes a program against Guile version 2.2.3, it will be compatible with some future version 2.2.7. We say in this case that 2.2 is the effective version, composed of the major and minor versions, in this case 2 and 2.

Users may install multiple effective versions of Guile, with each version’s headers, libraries, and Scheme files under their own directories. This provides the necessary stability guarantee for users, while also allowing Guile developers to evolve the language and its implementation.

However, parallel installability does have a down-side, in that users need to know which version of Guile to ask for, when they build against Guile. Guile solves this problem by installing a file to be read by the pkg-config utility, a tool to query installed packages by name. Guile encodes the version into its pkg-config name, so that users can ask for guile-2.0 or guile-2.2, as appropriate.

For effective version 2.2, for example, you would invoke pkg-config --cflags --libs guile-2.2 to get the compilation and linking flags necessary to link to version 2.2 of Guile. You would typically run pkg-config during the configuration phase of your program and use the obtained information in the Makefile.

Guile’s pkg-config file, guile-2.2.pc, defines additional useful variables:

sitedir The default directory where Guile looks for Scheme source and compiled files (see Section 4.7 [Installing Site Packages], page 57). Run pkg-config guile-2.2 --variable=sitedir to see its value. See Section 5.8.2 [Autoconf Macros], page 94, for more on how to use it from Autoconf.
extensiondir

The default directory where Guile looks for extensions—i.e., shared libraries providing additional features (see Section 6.21.4 [Modules and Extensions], page 426). Run `pkg-config guile-2.2 --variable=extensiondir` to see its value.

guile

guild

The absolute file name of the `guile` and `guild` commands\(^1\). Run `pkg-config guile-2.2 --variable=guile` or `--variable=guild` to see their value.

These variables allow users to deal with program name transformations that may be specified when configuring Guile with `--program-transform-name`, `--program-suffix`, or `--program-prefix` (see Section “Transformation Options” in GNU Autoconf Manual).

See the `pkg-config` man page, for more information, or its web site, [http://pkg-config.freedesktop.org/](http://pkg-config.freedesktop.org/). See Section 5.8 [Autoconf Support], page 94, for more on checking for Guile from within a `configure.ac` file.

5.2 Linking Programs With Guile

This section covers the mechanics of linking your program with Guile on a typical POSIX system.

The header file `<libguile.h>` provides declarations for all of Guile’s functions and constants. You should `#include` it at the head of any C source file that uses identifiers described in this manual. Once you’ve compiled your source files, you need to link them against the Guile object code library, `libguile`.

As noted in the previous section, `<libguile.h>` is not in the default search path for headers. The following command lines give respectively the C compilation and link flags needed to build programs using Guile 2.2:

```
pkg-config guile-2.2 --cflags
gpkg-config guile-2.2 --libs
```

5.2.1 Guile Initialization Functions

To initialize Guile, you can use one of several functions. The first, `scm_with_guile`, is the most portable way to initialize Guile. It will initialize Guile when necessary and then call a function that you can specify. Multiple threads can call `scm_with_guile` concurrently and it can also be called more than once in a given thread. The global state of Guile will survive from one call of `scm_with_guile` to the next. Your function is called from within `scm_with_guile` since the garbage collector of Guile needs to know where the stack of each thread is.

A second function, `scm_init_guile`, initializes Guile for the current thread. When it returns, you can use the Guile API in the current thread. This function employs some non-portable magic to learn about stack bounds and might thus not be available on all platforms.

One common way to use Guile is to write a set of C functions which perform some useful task, make them callable from Scheme, and then link the program with Guile. This yields

---

\(^1\) The `guile` and `guild` variables defined starting from Guile version 2.0.12.
a Scheme interpreter just like guile, but augmented with extra functions for some specific application — a special-purpose scripting language.

In this situation, the application should probably process its command-line arguments in the same manner as the stock Guile interpreter. To make that straightforward, Guile provides the scm_boot_guile and scm_shell function.

For more about these functions, see Section 6.4 [Initialization], page 101.

5.2.2 A Sample Guile Main Program

Here is simple-guile.c, source code for a main and an inner_main function that will produce a complete Guile interpreter.

```c
/* simple-guile.c --- Start Guile from C. */

#include <libguile.h>

static void
inner_main (void *closure, int argc, char **argv)
{
    /* preparation */
    scm_shell (argc, argv);
    /* after exit */
}

int
main (int argc, char **argv)
{
    scm_boot_guile (argc, argv, inner_main, 0);
    return 0; /* never reached, see inner_main */
}
```

The main function calls scm_boot_guile to initialize Guile, passing it inner_main. Once scm_boot_guile is ready, it invokes inner_main, which calls scm_shell to process the command-line arguments in the usual way.

5.2.3 Building the Example with Make

Here is a Makefile which you can use to compile the example program. It uses pkg-config to learn about the necessary compiler and linker flags.

```bash
# Use GCC, if you have it installed.
CC=gcc

# Tell the C compiler where to find <libguile.h>
CFLAGS=`pkg-config --cflags guile-2.2`

# Tell the linker what libraries to use and where to find them.
LIBS=`pkg-config --libs guile-2.2`

simple-guile: simple-guile.o
```

${\text{CC}}$ simple-guile.o ${\text{LIBS}}$ -o simple-guile

simple-guile.o: simple-guile.c
${\text{CC}}$ -c ${\text{CFLAGS}}$ simple-guile.c

5.2.4 Building the Example with Autoconf

If you are using the GNU Autoconf package to make your application more portable, Autoconf will settle many of the details in the Makefile automatically, making it much simpler and more portable; we recommend using Autoconf with Guile. Here is a configure.ac file for simple-guile that uses the standard PKG_CHECK_MODULES macro to check for Guile. Autoconf will process this file into a configure script. We recommend invoking Autoconf via the autoreconf utility.

AC_INIT(simple-guile.c)

# Find a C compiler.
AC_PROG_CC

# Check for Guile
PKG_CHECK_MODULES([GUILE], [guile-2.2])

# Generate a Makefile, based on the results.
AC_OUTPUT(Makefile)

Run autoreconf -vif to generate configure.

Here is a Makefile.in template, from which the configure script produces a Makefile customized for the host system:

# The configure script fills in these values.
CC=@CC@
CFLAGS=@GUILE_CFLAGS@
LIBS=@GUILE_LIBS@

simple-guile: simple-guile.o
${CC} simple-guile.o ${LIBS} -o simple-guile

simple-guile.o: simple-guile.c
${CC} -c ${CFLAGS} simple-guile.c

The developer should use Autoconf to generate the configure script from the configure.ac template, and distribute configure with the application. Here’s how a user might go about building the application:

$ ls
Makefile.in configure* configure.ac simple-guile.c
$ ./configure
checking for gcc... ccache gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether ccache gcc accepts -g... yes
checking for ccache gcc option to accept ISO C89... none needed
checking for pkg-config... /usr/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking for GUIL... yes
configure: creating ./config.status
config.status: creating Makefile
$ make
[...
$ ./simple-guile
guile> (+ 1 2 3)
6
guile> (getpwnam "jimb")
#("jimb" "83Z7d75W2tyJQ" 4008 10 "Jim Blandy" "/u/jimb"
   "/usr/local/bin/bash")
guile> (exit)
$

5.3 Linking Guile with Libraries

The previous section has briefly explained how to write programs that make use of an embedded Guile interpreter. But sometimes, all you want to do is make new primitive procedures and data types available to the Scheme programmer. Writing a new version of guile is inconvenient in this case and it would in fact make the life of the users of your new features needlessly hard.

For example, suppose that there is a program guile-db that is a version of Guile with additional features for accessing a database. People who want to write Scheme programs that use these features would have to use guile-db instead of the usual guile program. Now suppose that there is also a program guile-gtk that extends Guile with access to the popular Gtk+ toolkit for graphical user interfaces. People who want to write GUIs in Scheme would have to use guile-gtk. Now, what happens when you want to write a Scheme application that uses a GUI to let the user access a database? You would have to write a third program that incorporates both the database stuff and the GUI stuff. This might not be easy (because guile-gtk might be a quite obscure program, say) and taking this example further makes it easy to see that this approach can not work in practice.

It would have been much better if both the database features and the GUI feature had been provided as libraries that can just be linked with guile. Guile makes it easy to do just this, and we encourage you to make your extensions to Guile available as libraries whenever possible.

You write the new primitive procedures and data types in the normal fashion, and link them into a shared library instead of into a stand-alone program. The shared library can then be loaded dynamically by Guile.
5.3.1 A Sample Guile Extension

This section explains how to make the Bessel functions of the C library available to Scheme. First we need to write the appropriate glue code to convert the arguments and return values of the functions from Scheme to C and back. Additionally, we need a function that will add them to the set of Guile primitives. Because this is just an example, we will only implement this for the j0 function.

Consider the following file `bessel.c`.

```c
#include <math.h>
#include <libguile.h>

SCM j0_wrapper (SCM x)
{
    return scm_from_double (j0 (scm_to_double (x)));
}

void init_bessel ()
{
    scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

This C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux:

```
gcc 'pkg-config --cflags guile-2.2' \
    -shared -o libguile-bessel.so -fPIC bessel.c
```

For creating shared libraries portably, we recommend the use of GNU Libtool (see Section “Introduction” in GNU Libtool).

A shared library can be loaded into a running Guile process with the function `load-extension`. In addition to the name of the library to load, this function also expects the name of a function from that library that will be called to initialize it. For our example, we are going to call the function `init_bessel` which will make `j0_wrapper` available to Scheme programs with the name `j0`. Note that we do not specify a filename extension such as `.so` when invoking `load-extension`. The right extension for the host platform will be provided automatically.

```
(load-extension "libguile-bessel" "init_bessel")
(j0 2)
⇒ 0.223890779141236
```

For this to work, `load-extension` must be able to find `libguile-bessel`, of course. It will look in the places that are usual for your operating system, and it will additionally look into the directories listed in the `LTLIBRARY_PATH` environment variable.

To see how these Guile extensions via shared libraries relate to the module system, See Section 2.5.3 [Putting Extensions into Modules], page 12.

5.4 General concepts for using libguile

When you want to embed the Guile Scheme interpreter into your program or library, you need to link it against the `libguile` library (see Section 5.2 [Linking Programs With Guile], page 60). Once you have done this, your C code has access to a number of data types and
functions that can be used to invoke the interpreter, or make new functions that you have
written in C available to be called from Scheme code, among other things.

Scheme is different from C in a number of significant ways, and Guile tries to make the
advantages of Scheme available to C as well. Thus, in addition to a Scheme interpreter,
libguile also offers dynamic types, garbage collection, continuations, arithmetic on arbitrary
sized numbers, and other things.

The two fundamental concepts are dynamic types and garbage collection. You need to
understand how libguile offers them to C programs in order to use the rest of libguile. Also,
the more general control flow of Scheme caused by continuations needs to be dealt with.

Running asynchronous signal handlers and multi-threading is known to C code already,
but there are of course a few additional rules when using them together with libguile.

5.4.1 Dynamic Types

Scheme is a dynamically-typed language; this means that the system cannot, in general,
determine the type of a given expression at compile time. Types only become apparent at
run time. Variables do not have fixed types; a variable may hold a pair at one point, an
integer at the next, and a thousand-element vector later. Instead, values, not variables,
have fixed types.

In order to implement standard Scheme functions like pair? and string? and provide
garbage collection, the representation of every value must contain enough information to
accurately determine its type at run time. Often, Scheme systems also use this information
to determine whether a program has attempted to apply an operation to an inappropriately
typed value (such as taking the car of a string).

Because variables, pairs, and vectors may hold values of any type, Scheme implementa-
tions use a uniform representation for values — a single type large enough to hold either a
complete value or a pointer to a complete value, along with the necessary typing informa-
tion.

In Guile, this uniform representation of all Scheme values is the C type SCM. This is
an opaque type and its size is typically equivalent to that of a pointer to void. Thus, SCM
values can be passed around efficiently and they take up reasonably little storage on their
own.

The most important rule is: You never access a SCM value directly; you only pass it to
functions or macros defined in libguile.

As an obvious example, although a SCM variable can contain integers, you can of course
not compute the sum of two SCM values by adding them with the C + operator. You must
use the libguile function scm_sum.

Less obvious and therefore more important to keep in mind is that you also cannot
directly test SCM values for trueness. In Scheme, the value #f is considered false and of
course a SCM variable can represent that value. But there is no guarantee that the SCM
representation of #f looks false to C code as well. You need to use scm_is_true or scm_
is_false to test a SCM value for trueness or falseness, respectively.

You also can not directly compare two SCM values to find out whether they are identical
(that is, whether they are eq? in Scheme terms). You need to use scm_is_eq for this.

The one exception is that you can directly assign a SCM value to a SCM variable by using
the C = operator.
The following (contrived) example shows how to do it right. It implements a function of two arguments \( (a \text{ and } \text{flag}) \) that returns \( a+1 \) if \( \text{flag} \) is true, else it returns \( a \) unchanged.

```scm
my_incrementing_function (SCM a, SCM flag)
{
    SCM result;
    if (scm_is_true (flag))
        result = scm_sum (a, scm_from_int (1));
    else
        result = a;

    return result;
}
```

Often, you need to convert between SCM values and appropriate C values. For example, we needed to convert the integer \( 1 \) to its SCM representation in order to add it to \( a \). Libguile provides many function to do these conversions, both from C to SCM and from SCM to C.

The conversion functions follow a common naming pattern: those that make a SCM value from a C value have names of the form `scm_from_type (...)` and those that convert a SCM value to a C value use the form `scm_to_type (...)`.

However, it is best to avoid converting values when you can. When you must combine C values and SCM values in a computation, it is often better to convert the C values to SCM values and do the computation by using libguile functions than to the other way around (converting SCM to C and doing the computation some other way).

As a simple example, consider this version of `my_incrementing_function` from above:

```scm
my_other_incrementing_function (SCM a, SCM flag)
{
    int result;
    if (scm_is_true (flag))
        result = scm_to_int (a) + 1;
    else
        result = scm_to_int (a);

    return scm_from_int (result);
}
```

This version is much less general than the original one: it will only work for values \( A \) that can fit into a `int`. The original function will work for all values that Guile can represent and that `scm_sum` can understand, including integers bigger than `long long`, floating point numbers, complex numbers, and new numerical types that have been added to Guile by third-party libraries.

Also, computing with SCM is not necessarily inefficient. Small integers will be encoded directly in the SCM value, for example, and do not need any additional memory on the heap. See Section 9.2 [Data Representation], page 803, to find out the details.
Some special SCM values are available to C code without needing to convert them from C values:

<table>
<thead>
<tr>
<th>Scheme value</th>
<th>C representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>#f</td>
<td>SCM_BOOL_F</td>
</tr>
<tr>
<td>#t</td>
<td>SCM_BOOL_T</td>
</tr>
<tr>
<td>()</td>
<td>SCM_EOL</td>
</tr>
</tbody>
</table>

In addition to SCM, Guile also defines the related type scm_t_bits. This is an unsigned integral type of sufficient size to hold all information that is directly contained in a SCM value. The scm_t_bits type is used internally by Guile to do all the bit twiddling explained in Section 9.2 [Data Representation], page 803, but you will encounter it occasionally in low-level user code as well.

### 5.4.2 Garbage Collection

As explained above, the SCM type can represent all Scheme values. Some values fit entirely into a SCM value (such as small integers), but other values require additional storage in the heap (such as strings and vectors). This additional storage is managed automatically by Guile. You don’t need to explicitly deallocate it when a SCM value is no longer used.

Two things must be guaranteed so that Guile is able to manage the storage automatically: it must know about all blocks of memory that have ever been allocated for Scheme values, and it must know about all Scheme values that are still being used. Given this knowledge, Guile can periodically free all blocks that have been allocated but are not used by any active Scheme values. This activity is called garbage collection.

Guile’s garbage collector will automatically discover references to SCM objects that originate in global variables, static data sections, function arguments or local variables on the C and Scheme stacks, and values in machine registers. Other references to SCM objects, such as those in other random data structures in the C heap that contain fields of type SCM, can be made visible to the garbage collector by calling the functions scm_gc_protect_object or scm_permanent_object. Collectively, these values form the “root set” of garbage collection; any value on the heap that is referenced directly or indirectly by a member of the root set is preserved, and all other objects are eligible for reclamation.

In Guile, garbage collection has two logical phases: the mark phase, in which the collector discovers the set of all live objects, and the sweep phase, in which the collector reclaims the resources associated with dead objects. The mark phase pauses the program and traces all SCM object references, starting with the root set. The sweep phase actually runs concurrently with the main program, incrementally reclaiming memory as needed by allocation.

In the mark phase, the garbage collector traces the Scheme stack and heap precisely. Because the Scheme stack and heap are managed by Guile, Guile can know precisely where in those data structures it might find references to other heap objects. This is not the case, unfortunately, for pointers on the C stack and static data segment. Instead of requiring the user to inform Guile about all variables in C that might point to heap objects, Guile traces the C stack and static data segment conservatively. That is to say, Guile just treats every word on the C stack and every C global variable as a potential reference in to the
Scheme heap\(^2\). Any value that looks like a pointer to a GC-managed object is treated as such, whether it actually is a reference or not. Thus, scanning the C stack and static data segment is guaranteed to find all actual references, but it might also find words that only accidentally look like references. These “false positives” might keep SCM objects alive that would otherwise be considered dead. While this might waste memory, keeping an object around longer than it strictly needs to is harmless. This is why this technique is called “conservative garbage collection”. In practice, the wasted memory seems to be no problem, as the static C root set is almost always finite and small, given that the Scheme stack is separate from the C stack.

The stack of every thread is scanned in this way and the registers of the CPU and all other memory locations where local variables or function parameters might show up are included in this scan as well.

The consequence of the conservative scanning is that you can just declare local variables and function parameters of type SCM and be sure that the garbage collector will not free the corresponding objects.

However, a local variable or function parameter is only protected as long as it is really on the stack (or in some register). As an optimization, the C compiler might reuse its location for some other value and the SCM object would no longer be protected. Normally, this leads to exactly the right behavior: the compiler will only overwrite a reference when it is no longer needed and thus the object becomes unprotected precisely when the reference disappears, just as wanted.

There are situations, however, where a SCM object needs to be around longer than its reference from a local variable or function parameter. This happens, for example, when you retrieve some pointer from a foreign object and work with that pointer directly. The reference to the SCM foreign object might be dead after the pointer has been retrieved, but the pointer itself (and the memory pointed to) is still in use and thus the foreign object must be protected. The compiler does not know about this connection and might overwrite the SCM reference too early.

To get around this problem, you can use \texttt{scm\textunderscore remember\textunderscore upto\textunderscore here\_1} and its cousins. It will keep the compiler from overwriting the reference. See Section 5.5.4 [Foreign Object Memory Management], page 77.

### 5.4.3 Control Flow

Scheme has a more general view of program flow than C, both locally and non-locally.

Controlling the local flow of control involves things like gotos, loops, calling functions and returning from them. Non-local control flow refers to situations where the program jumps across one or more levels of function activations without using the normal call or return operations.

The primitive means of C for local control flow is the \texttt{goto} statement, together with \texttt{if}. Loops done with \texttt{for}, \texttt{while} or \texttt{do} could in principle be rewritten with just \texttt{goto} and \texttt{if}. In Scheme, the primitive means for local control flow is the \textit{function call} (together with \texttt{if}).

\(^2\) Note that Guile does not scan the C heap for references, so a reference to a SCM object from a memory segment allocated with \texttt{malloc} will have to use some other means to keep the SCM object alive. See Section 6.19.1 [Garbage Collection Functions], page 401.
Thus, the repetition of some computation in a loop is ultimately implemented by a function that calls itself, that is, by recursion.

This approach is theoretically very powerful since it is easier to reason formally about recursion than about gotos. In C, using recursion exclusively would not be practical, though, since it would eat up the stack very quickly. In Scheme, however, it is practical: function calls that appear in a tail position do not use any additional stack space (see Section 3.3.2 [Tail Calls], page 24).

A function call is in a tail position when it is the last thing the calling function does. The value returned by the called function is immediately returned from the calling function. In the following example, the call to bar-1 is in a tail position, while the call to bar-2 is not. (The call to 1- in foo-2 is in a tail position, though.)

\[
\begin{align*}
\text{(define (foo-1 x)} \\
& \quad (\text{bar-1 (1- x)}))
\end{align*}
\]

\[
\begin{align*}
\text{(define (foo-2 x)} \\
& \quad (1- (\text{bar-2 x})))
\end{align*}
\]

Thus, when you take care to recurse only in tail positions, the recursion will only use constant stack space and will be as good as a loop constructed from gotos.

Scheme offers a few syntactic abstractions (do and named let) that make writing loops slightly easier.

But only Scheme functions can call other functions in a tail position: C functions cannot. This matters when you have, say, two functions that call each other recursively to form a common loop. The following (unrealistic) example shows how one might go about determining whether a non-negative integer \( n \) is even or odd.

\[
\begin{align*}
\text{(define (my-even? n)} \\
& \quad (\text{cond ((zero? n) #t)} \\
& \quad \quad (\text{else (my-odd? (1- n))}))
\end{align*}
\]

\[
\begin{align*}
\text{(define (my-odd? n)} \\
& \quad (\text{cond ((zero? n) #f)} \\
& \quad \quad (\text{else (my-even? (1- n))}))
\end{align*}
\]

Because the calls to my-even? and my-odd? are in tail positions, these two procedures can be applied to arbitrary large integers without overflowing the stack. (They will still take a lot of time, of course.)

However, when one or both of the two procedures would be rewritten in C, it could no longer call its companion in a tail position (since C does not have this concept). You might need to take this consideration into account when deciding which parts of your program to write in Scheme and which in C.

In addition to calling functions and returning from them, a Scheme program can also exit non-locally from a function so that the control flow returns directly to an outer level. This means that some functions might not return at all.

Even more, it is not only possible to jump to some outer level of control, a Scheme program can also jump back into the middle of a function that has already exited. This might cause some functions to return more than once.
In general, these non-local jumps are done by invoking \textit{continuations} that have previously been captured using \texttt{call-with-current-continuation}. Guile also offers a slightly restricted set of functions, \texttt{catch} and \texttt{throw}, that can only be used for non-local exits. This restriction makes them more efficient. Error reporting (with the function \texttt{error}) is implemented by invoking \texttt{throw}, for example. The functions \texttt{catch} and \texttt{throw} belong to the topic of \textit{exceptions}.

Since Scheme functions can call C functions and vice versa, C code can experience the more general control flow of Scheme as well. It is possible that a C function will not return at all, or will return more than once. While C does offer \texttt{setjmp} and \texttt{longjmp} for non-local exits, it is still an unusual thing for C code. In contrast, non-local exits are very common in Scheme, mostly to report errors.

You need to be prepared for the non-local jumps in the control flow whenever you use a function from \texttt{libguile}: it is best to assume that any \texttt{libguile} function might signal an error or run a pending signal handler (which in turn can do arbitrary things).

It is often necessary to take cleanup actions when the control leaves a function non-locally. Also, when the control returns non-locally, some setup actions might be called for. For example, the Scheme function \texttt{with-output-to-port} needs to modify the global state so that \texttt{current-output-port} returns the port passed to \texttt{with-output-to-port}. The global output port needs to be reset to its previous value when \texttt{with-output-to-port} returns normally or when it is exited non-locally. Likewise, the port needs to be set again when control enters non-locally.

Scheme code can use the \texttt{dynamic-wind} function to arrange for the setting and resetting of the global state. C code can use the corresponding \texttt{scm_internal_dynamic_wind} function, or a \texttt{scmDynwindBegin/scmDynwindEnd} pair together with suitable 'dynwind actions' (see Section 6.13.10 [Dynamic Wind], page 317).

Instead of coping with non-local control flow, you can also prevent it by erecting a \textit{continuation barrier}. See Section 6.13.14 [Continuation Barriers], page 328. The function \texttt{scmCWithContinuationBarrier}, for example, is guaranteed to return exactly once.

### 5.4.4 Asynchronous Signals

You can not call libguile functions from handlers for POSIX signals, but you can register Scheme handlers for POSIX signals such as \texttt{SIGINT}. These handlers do not run during the actual signal delivery. Instead, they are run when the program (more precisely, the thread that the handler has been registered for) reaches the next \textit{safe point}.

The libguile functions themselves have many such safe points. Consequently, you must be prepared for arbitrary actions anytime you call a libguile function. For example, even \texttt{scmCons} can contain a safe point and when a signal handler is pending for your thread, calling \texttt{scmCons} will run this handler and anything might happen, including a non-local exit although \texttt{scmCons} would not ordinarily do such a thing on its own.

If you do not want to allow the running of asynchronous signal handlers, you can block them temporarily with \texttt{scmDynwindBlockAsyncs}, for example. See Section 6.22.3 [Asyncs], page 439.

Since signal handling in Guile relies on safe points, you need to make sure that your functions do offer enough of them. Normally, calling libguile functions in the normal course of action is all that is needed. But when a thread might spent a long time in a code section...
that calls no libguile function, it is good to include explicit safe points. This can allow the
user to interrupt your code with C-c, for example.

You can do this with the macro SCM_TICK. This macro is syntactically a statement. That is, you could use it like this:

```c
while (1)
{
    SCM_TICK;
    do_some_work ();
}
```

Frequent execution of a safe point is even more important in multi threaded programs, See Section 5.4.5 [Multi-Threading], page 71.

### 5.4.5 Multi-Threading

Guile can be used in multi-threaded programs just as well as in single-threaded ones.

Each thread that wants to use functions from libguile must put itself into guile mode and must then follow a few rules. If it doesn’t want to honor these rules in certain situations, a thread can temporarily leave guile mode (but can no longer use libguile functions during that time, of course).

Threads enter guile mode by calling `scm_with_guile`, `scm_boot_guile`, or `scm_init_guile`. As explained in the reference documentation for these functions, Guile will then learn about the stack bounds of the thread and can protect the SCM values that are stored in local variables. When a thread puts itself into guile mode for the first time, it gets a Scheme representation and is listed by all-threads, for example.

Threads in guile mode can block (e.g., do blocking I/O) without causing any problems\(^3\); temporarily leaving guile mode with `scm_without_guile` before blocking slightly improves GC performance, though. For some common blocking operations, Guile provides convenience functions. For example, if you want to lock a pthread mutex while in guile mode, you might want to use `scm_pthread_mutex_lock` which is just like `pthread_mutex_lock` except that it leaves guile mode while blocking.

All libguile functions are (intended to be) robust in the face of multiple threads using them concurrently. This means that there is no risk of the internal data structures of libguile becoming corrupted in such a way that the process crashes.

A program might still produce nonsensical results, though. Taking hashtables as an example, Guile guarantees that you can use them from multiple threads concurrently and a hashtable will always remain a valid hashtable and Guile will not crash when you access it. It does not guarantee, however, that inserting into it concurrently from two threads will give useful results: only one insertion might actually happen, none might happen, or the table might in general be modified in a totally arbitrary manner. (It will still be a valid hashtable, but not the one that you might have expected.) Guile might also signal an error when it detects a harmful race condition.

Thus, you need to put in additional synchronizations when multiple threads want to use a single hashtable, or any other mutable Scheme object.

---

\(^3\) In Guile 1.8, a thread blocking in guile mode would prevent garbage collection to occur. Thus, threads had to leave guile mode whenever they could block. This is no longer needed with Guile 2.x.
When writing C code for use with libguile, you should try to make it robust as well. An example that converts a list into a vector will help to illustrate. Here is a correct version:

```c
SCM
my_list_to_vector (SCM list)
{
    SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
    size_t len, i;

    len = scm_c_vector_length (vector);
    i = 0;
    while (i < len && scm_is_pair (list))
    {
        scm_c_vector_set_x (vector, i, scm_car (list));
        list = scm_cdr (list);
        i++;
    }

    return vector;
}
```

The first thing to note is that storing into a SCM location concurrently from multiple threads is guaranteed to be robust: you don’t know which value wins but it will in any case be a valid SCM value.

But there is no guarantee that the list referenced by list is not modified in another thread while the loop iterates over it. Thus, while copying its elements into the vector, the list might get longer or shorter. For this reason, the loop must check both that it doesn’t overrun the vector and that it doesn’t overrun the list. Otherwise, `scm_c_vector_set_x` would raise an error if the index is out of range, and `scm_car` and `scm_cdr` would raise an error if the value is not a pair.

It is safe to use `scm_car` and `scm_cdr` on the local variable list once it is known that the variable contains a pair. The contents of the pair might change spontaneously, but it will always stay a valid pair (and a local variable will of course not spontaneously point to a different Scheme object).

Likewise, a vector such as the one returned by `scm_make_vector` is guaranteed to always stay the same length so that it is safe to only use `scm_c_vector_length` once and store the result. (In the example, vector is safe anyway since it is a fresh object that no other thread can possibly know about until it is returned from `my_list_to_vector`.)

Of course the behavior of `my_list_to_vector` is suboptimal when list does indeed get asynchronously lengthened or shortened in another thread. But it is robust: it will always return a valid vector. That vector might be shorter than expected, or its last elements might be unspecified, but it is a valid vector and if a program wants to rule out these cases, it must avoid modifying the list asynchronously.

Here is another version that is also correct:

```c
SCM
my_pedantic_list_to_vector (SCM list)
{
```
SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
size_t len, i;

len = scm_c_vector_length (vector);
i = 0;
while (i < len)
{
    scm_c_vector_set_x (vector, i, scm_car (list));
    list = scm_cdr (list);
    i++;
}

return vector;
}

This version relies on the error-checking behavior of scm_car and scm_cdr. When the list is shortened (that is, when list holds a non-pair), scm_car will throw an error. This might be preferable to just returning a half-initialized vector.

The API for accessing vectors and arrays of various kinds from C takes a slightly different approach to thread-robustness. In order to get at the raw memory that stores the elements of an array, you need to reserve that array as long as you need the raw memory. During the time an array is reserved, its elements can still spontaneously change their values, but the memory itself and other things like the size of the array are guaranteed to stay fixed. Any operation that would change these parameters of an array that is currently reserved will signal an error. In order to avoid these errors, a program should of course put suitable synchronization mechanisms in place. As you can see, Guile itself is again only concerned about robustness, not about correctness: without proper synchronization, your program will likely not be correct, but the worst consequence is an error message.

Real thread-safety often requires that a critical section of code is executed in a certain restricted manner. A common requirement is that the code section is not entered a second time when it is already being executed. Locking a mutex while in that section ensures that no other thread will start executing it, blocking asyncs ensures that no asynchronous code enters the section again from the current thread, and the error checking of Guile mutexes guarantees that an error is signalled when the current thread accidentally reenters the critical section via recursive function calls.

Guile provides two mechanisms to support critical sections as outlined above. You can either use the macros SCM_CRITICAL_SECTION_START and SCM_CRITICAL_SECTION_END for very simple sections; or use a dynwind context together with a call to scm_dynwind_critical_section.

The macros only work reliably for critical sections that are guaranteed to not cause a non-local exit. They also do not detect an accidental reentry by the current thread. Thus, you should probably only use them to delimit critical sections that do not contain calls to libguile functions or to other external functions that might do complicated things.

The function scm_dynwind_critical_section, on the other hand, will correctly deal with non-local exits because it requires a dynwind context. Also, by using a separate mutex for each critical section, it can detect accidental reentries.
5.5 Defining New Foreign Object Types

The foreign object type facility is Guile’s mechanism for importing object and types from C or other languages into Guile’s system. If you have a C struct foo type, for example, you can define a corresponding Guile foreign object type that allows Scheme code to handle struct foo * objects.

To define a new foreign object type, the programmer provides Guile with some essential information about the type — what its name is, how many fields it has, and its finalizer (if any) — and Guile allocates a fresh type for it. Foreign objects can be accessed from Scheme or from C.

5.5.1 Defining Foreign Object Types

To create a new foreign object type from C, call scm_make_foreign_object_type. It returns a value of type SCM which identifies the new type.

Here is how one might declare a new type representing eight-bit gray-scale images:

```c
#include <libguile.h>

struct image {
  int width, height;
  char *pixels;

  /* The name of this image */
  SCM name;

  /* A function to call when this image is modified, e.g., to update the screen,
   * or SCM_BOOL_F if no action necessary */
  SCM update_func;
};

static SCM image_type;

void init_image_type (void)
{
  SCM name, slots;
  scm_t_struct_finalize finalizer;

  name = scm_from_utf8_symbol ("image");
  slots = scm_list_1 (scm_from_utf8_symbol ("data"));
  finalizer = NULL;

  image_type =
    scm_make_foreign_object_type (name, slots, finalizer);
}
```
The result is an initialized `image_type` value that identifies the new foreign object type. The next section describes how to create foreign objects and how to access their slots.

### 5.5.2 Creating Foreign Objects

Foreign objects contain zero or more “slots” of data. A slot can hold a pointer, an integer that fits into a `size_t` or `ssize_t`, or a `SCM` value.

All objects of a given foreign type have the same number of slots. In the example from the previous section, the `image` type has one slot, because the slots list passed to `scm_make_foreign_object_type` is of length one. (The actual names given to slots are unimportant for most users of the C interface, but can be used on the Scheme side for introspection on the foreign object.)

To construct a foreign object and initialize its first slot, call `scm_make_foreign_object_1` (`type, first_slot_value`). There are similarly named constructors for initializing 0, 1, 2, or 3 slots, or initializing `n` slots via an array. See Section 6.7 [Foreign Objects], page 243, for full details. Any fields that are not explicitly initialized are set to 0.

To get or set the value of a slot by index, you can use the `scm_foreign_object_ref` and `scm_foreign_object_set_x` functions. These functions take and return values as `void *` pointers; there are corresponding convenience procedures like `_signed_ref`, `_unsigned_set_x` and so on for dealing with slots as signed or unsigned integers.

Foreign objects fields that are pointers can be tricky to manage. If possible, it is best that all memory that is referenced by a foreign object be managed by the garbage collector. That way, the GC can automatically ensure that memory is accessible when it is needed, and freed when it becomes inaccessible. If this is not the case for your program – for example, if you are exposing an object to Scheme that was allocated by some other, Guile-unaware part of your program – then you will probably need to implement a finalizer. See Section 5.5.4 [Foreign Object Memory Management], page 77, for more.

Continuing the example from the previous section, if the global variable `image_type` contains the type returned by `scm_make_foreign_object_type`, here is how we could construct a foreign object whose “data” field contains a pointer to a freshly allocated `struct` image:

```c
SCM
make_image (SCM name, SCM s_width, SCM s_height)
{
    struct image *image;
    int width = scm_to_int (s_width);
    int height = scm_to_int (s_height);

    /* Allocate the ‘struct image’. Because we
     * use scm_gc_malloc, this memory block will
     * be automatically reclaimed when it becomes
     * inaccessible, and its members will be traced
     * by the garbage collector. */
    image = (struct image *)
        scm_gc_malloc (sizeof (struct image), "image");
```
image->width = width;
image->height = height;

/* Allocating the pixels with
   scm_gc_malloc_pointerless means that the
   pixels data is collectable by GC, but
   that GC shouldn't spend time tracing its
   contents for nested pointers because there
   aren't any. */
image->pixels =
   scm_gc_malloc_pointerless (width * height, "image pixels");

image->name = name;
image->update_func = SCM_BOOL_F;

/* Now wrap the struct image* in a new foreign
   object, and return that object. */
return scm_make_foreign_object_1 (image_type, image);
}

We use scm_gc_malloc_pointerless for the pixel buffer to tell the garbage collector
not to scan it for pointers. Calls to scm_gc_malloc, scm_make_foreign_object_1, and
scm_gc_malloc_pointerless raise an exception in out-of-memory conditions; the garbage
collector is able to reclaim previously allocated memory if that happens.

5.5.3 Type Checking of Foreign Objects

Functions that operate on foreign objects should check that the passed SCM value indeed is
of the correct type before accessing its data. They can do this with scm_assert_foreign_
object_type.

For example, here is a simple function that operates on an image object, and checks the
type of its argument.

SCM
clear_image (SCM image_obj)
{
   int area;
   struct image *image;

   scm_assert_foreign_object_type (image_type, image_obj);

   image = scm_foreign_object_ref (image_obj, 0);
   area = image->width * image->height;
   memset (image->pixels, 0, area);

   /* Invoke the image’s update function. */
   if (scm_is_true (image->update_func))
      scm_call_0 (image->update_func);
5.5.4 Foreign Object Memory Management

Once a foreign object has been released to the tender mercies of the Scheme system, it must be prepared to survive garbage collection. In the example above, all the memory associated with the foreign object is managed by the garbage collector because we used the `scm_gc_` allocation functions. Thus, no special care must be taken: the garbage collector automatically scans them and reclaims any unused memory.

However, when data associated with a foreign object is managed in some other way—e.g., `malloc`'d memory or file descriptors—it is possible to specify a finalizer function to release those resources when the foreign object is reclaimed.

As discussed in see Section 5.4.2 [Garbage Collection], page 67, Guile’s garbage collector will reclaim inaccessible memory as needed. This reclamation process runs concurrently with the main program. When Guile analyzes the heap and determines that an object’s memory can be reclaimed, that memory is put on a “free list” of objects that can be reclaimed. Usually that’s the end of it—the object is available for immediate re-use. However some objects can have “finalizers” associated with them—functions that are called on reclaimable objects to effect any external cleanup actions.

Finalizers are tricky business and it is best to avoid them. They can be invoked at unexpected times, or not at all—for example, they are not invoked on process exit. They don’t help the garbage collector do its job; in fact, they are a hindrance. Furthermore, they perturb the garbage collector’s internal accounting. The GC decides to scan the heap when it thinks that it is necessary, after some amount of allocation. Finalizable objects almost always represent an amount of allocation that is invisible to the garbage collector. The effect can be that the actual resource usage of a system with finalizable objects is higher than what the GC thinks it should be.

All those caveats aside, some foreign object types will need finalizers. For example, if we had a foreign object type that wrapped file descriptors—and we aren’t suggesting this, as Guile already has ports —then you might define the type like this:

```c
static SCM file_type;

static void
finalize_file (SCM file)
{
    int fd = scm_foreign_object_signed_ref (file, 0);
    if (fd >= 0)
    {
        scm_foreign_object_signed_set_x (file, 0, -1);
        close (fd);
    }
}

static void
init_file_type (void)
{
    return SCM_UNSPECIFIED;
}
```
SCM name, slots;
scm_t_struct_finalize finalizer;

name = scm_from_utf8_symbol ("file");
slots = scm_list_1 (scm_from_utf8_symbol ("fd"));
finalizer = finalize_file;

image_type =
    scm_make_foreign_object_type (name, slots, finalizer);
}

static SCM
make_file (int fd)
{
    return scm_make_foreign_object_1 (file_type, (void *) fd);
}

Note that the finalizer may be invoked in ways and at times you might not expect. In particular, if the user’s Guile is built with support for threads, the finalizer may be called from any thread that is running Guile. In Guile 2.0, finalizers are invoked via “asyncs”, which interleaves them with running Scheme code; see Section 6.22.3 [Asyncs], page 439. In Guile 2.2 there will be a dedicated finalization thread, to ensure that the finalization doesn’t run within the critical section of any other thread known to Guile.

In either case, finalizers run concurrently with the main program, and so they need to be async-safe and thread-safe. If for some reason this is impossible, perhaps because you are embedding Guile in some application that is not itself thread-safe, you have a few options. One is to use guardians instead of finalizers, and arrange to pump the guardians for finalizable objects. See Section 6.19.4 [Guardians], page 405, for more information. The other option is to disable automatic finalization entirely, and arrange to call scm_run_finalizers () at appropriate points. See Section 6.7 [Foreign Objects], page 243, for more on these interfaces.

Finalizers are allowed to allocate memory, access GC-managed memory, and in general can do anything any Guile user code can do. This was not the case in Guile 1.8, where finalizers were much more restricted. In particular, in Guile 2.0, finalizers can resuscitate objects. We do not recommend that users avail themselves of this possibility, however, as a resuscitated object can re-expose other finalizable objects that have been already finalized back to Scheme. These objects will not be finalized again, but they could cause use-after-free problems to code that handles objects of that particular foreign object type. To guard against this possibility, robust finalization routines should clear state from the foreign object, as in the above free_file example.

One final caveat. Foreign object finalizers are associated with the lifetime of a foreign object, not of its fields. If you access a field of a finalizable foreign object, and do not arrange to keep a reference on the foreign object itself, it could be that the outer foreign object gets finalized while you are working with its field.

For example, consider a procedure to read some data from a file, from our example above.

SCM
read_bytes (SCM file, SCM n)
{
  int fd;
  SCM buf;
  size_t len, pos;

  scm_assert_foreign_object_type (file_type, file);

  fd = scm_foreign_object_signed_ref (file, 0);
  if (fd < 0)
    scm_wrong_type_arg_msg ("read-bytes", SCM_ARG1, file, "open file");

  len = scm_to_size_t (n);
  SCM buf = scm_c_make_bytevector (scm_to_size_t (n));

  pos = 0;
  while (pos < len)
  {
    char *bytes = SCM_BYTEVECTOR_CONTENTS (buf);
    ssize_t count = read (fd, bytes + pos, len - pos);
    if (count < 0)
      scm_syserror ("read-bytes");
    if (count == 0)
      break;
    pos += count;
  }

  scm_remember_upto_here_1 (file);

  return scm_values (scm_list_2 (buf, scm_from_size_t (pos)));
}

After the prelude, only the fd value is used and the C compiler has no reason to keep the file object around. If scm_c_make_bytevector results in a garbage collection, file might not be on the stack or anywhere else and could be finalized, leaving read to read a closed (or, in a multi-threaded program, possibly re-used) file descriptor. The use of scm_remember_upto_here_1 prevents this, by creating a reference to file after all data accesses. See Section 6.19.1 [Garbage Collection Functions], page 401.

scm_remember_upto_here_1 is only needed on finalizable objects, because garbage collection of other values is invisible to the program – it happens when needed, and is not observable. But if you can, save yourself the headache and build your program in such a way that it doesn't need finalization.
5.5.5 Foreign Objects and Scheme

It is also possible to create foreign objects and object types from Scheme, and to access fields of foreign objects from Scheme. For example, the file example from the last section could be equivalently expressed as:

```scheme
(define-module (my-file)
  #:use-module (system foreign-object)
  #:use-module ((oop goops) #:select (make))
  #:export (make-file))

(define (finalize-file file)
  (let ((fd (struct-ref file 0)))
    (unless (< fd 0)
      (struct-set! file 0 -1)
      (close-fdes fd)))))

(define <file>
  (make-foreign-object-type '<file> '(fd)
    #:finalizer finalize-file))

(define (make-file fd)
  (make <file> #:fd fd))

Here we see that the result of make-foreign-object-type, which is the equivalent of scm_make_foreign_object_type, is a struct vtable. See Section 6.6.18.1 [Vtables], page 223, for more information. To instantiate the foreign object, which is really a Guile struct, we use make. (We could have used make-struct/no-tail, but as an implementation detail, finalizers are attached in the initialize method called by make). To access the fields, we use struct-ref and struct-set!. See Section 6.6.18.2 [Structure Basics], page 223.

There is a convenience syntax, define-foreign-object-type, that defines a type along with a constructor, and getters for the fields. An appropriate invocation of define-foreign-object-type for the file object type could look like this:

```scheme
(use-modules (system foreign-object))

(define-foreign-object-type <file>
  make-file
  (fd)
  #:finalizer finalize-file)
```

This defines the <file> type with one field, a make-file constructor, and a getter for the fd field, bound to fd.

Foreign object types are not only vtables but are actually GOOPS classes, as hinted at above. See Chapter 8 [GOOPS], page 759, for more on Guile’s object-oriented programming system. Thus one can define print and equality methods using GOOPS:

```scheme
(use-modules (oop goops))

(define-method (write (file <file>)) port)
```
Chapter 5: Programming in C

81

;; Assuming existence of the ‘fd’ getter
(format port "#<<file> "a" (fd file)))

(define-method (equal? (a <file>) (b <file>))
  (eqv? (fd a) (fd b)))

One can even sub-class foreign types.

(define-class <named-file> (<file>)
  (name #:init-keyword #:name #:init-value #f #:accessor name))

The question arises of how to construct these values, given that make-file returns a
plain old <file> object. It turns out that you can use the GOOPS construction interface,
where every field of the foreign object has an associated initialization keyword argument.

(define* (my-open-file name #:optional (flags O_RDONLY))
  (make <named-file> #:fd (open-fdes name flags) #:name name))

(define-method (write (file <named-file>) port)
  (format port "#<<file> "s "a" (name file) (fd file)))

See Section 6.7 [Foreign Objects], page 243, for full documentation on the Scheme inter-
face to foreign objects. See Chapter 8 [GOOPS], page 759, for more on GOOPS.

As a final note, you might wonder how this system supports encapsulation of sensitive
values. First, we have to recognize that some facilities are essentially unsafe and have global
scope. For example, in C, the integrity and confidentiality of a part of a program is at the
mercy of every other part of that program – because any part of the program can read and
write anything in its address space. At the same time, principled access to structured data
is organized in C on lexical boundaries; if you don’t expose accessors for your object, you
trust other parts of the program not to work around that barrier.

The situation is not dissimilar in Scheme. Although Scheme’s unsafe constructs are
fewer in number than in C, they do exist. The (system foreign) module can be used to
violate confidentiality and integrity, and shouldn’t be exposed to untrusted code. Although
struct-ref and struct-set! are less unsafe, they still have a cross-cutting capability of
drilling through abstractions. Performing a struct-set! on a foreign object slot could
cause unsafe foreign code to crash. Ultimately, structures in Scheme are capabilities for
abstraction, and not abstractions themselves.

That leaves us with the lexical capabilities, like constructors and accessors. Here is
where encapsulation lies: the practical degree to which the innards of your foreign objects
are exposed is the degree to which their accessors are lexically available in user code. If you
want to allow users to reference fields of your foreign object, provide them with a getter.
Otherwise you should assume that the only access to your object may come from your code,
which has the relevant authority, or via code with access to cross-cutting struct-ref and
such, which also has the cross-cutting authority.

5.6 Function Snarfing

When writing C code for use with Guile, you typically define a set of C functions, and then
make some of them visible to the Scheme world by calling scm_c_define_gsubr or related
functions. If you have many functions to publish, it can sometimes be annoying to keep the
list of calls to scm_c_define_gsubr in sync with the list of function definitions.
Guile provides the `guile-snarf` program to manage this problem. Using this tool, you can keep all the information needed to define the function alongside the function definition itself; `guile-snarf` will extract this information from your source code, and automatically generate a file of calls to `scm_c_define_gsubr` which you can `#include` into an initialization function.

The snarfing mechanism works for many kinds of initialization actions, not just for collecting calls to `scm_c_define_gsubr`. For a full list of what can be done, See Section 6.5 [Snarfing Macros], page 102.

The `guile-snarf` program is invoked like this:

```
guile-snarf [-o outfile] [cpp-args ...]
```

This command will extract initialization actions to `outfile`. When no `outfile` has been specified or when `outfile` is `-`, standard output will be used. The C preprocessor is called with `cpp-args` (which usually include an input file) and the output is filtered to extract the initialization actions.

If there are errors during processing, `outfile` is deleted and the program exits with non-zero status.

During snarfing, the pre-processor macro `SCM_MAGIC_SNARFER` is defined. You could use this to avoid including snarfer output files that don’t yet exist by writing code like this:

```
#ifndef SCM_MAGIC_SNARFER
#include "foo.x"
#endif
```

Here is how you might define the Scheme function `clear-image`, implemented by the C function `clear_image`:

```
#include <libguile.h>

SCM_DEFINE (clear_image, "clear-image", 1, 0, 0,
    (SCM image),
    "Clear the image.")
{
    /* C code to clear the image in image... */
}
```

```
void
init_image_type ()
{
    #include "image-type.x"
}
```

The `SCM_DEFINE` declaration says that the C function `clear_image` implements a Scheme function called `clear-image`, which takes one required argument (of type SCM and named `image`), no optional arguments, and no rest argument. The string "Clear the image." provides a short help text for the function, it is called a docstring.

`SCM_DEFINE` macro also defines a static array of characters initialized to the Scheme name of the function. In this case, `s_clear_image` is set to the C string, "clear-image". You might want to use this symbol when generating error messages.
Assuming the text above lives in a file named image-type.c, you will need to execute the following command to prepare this file for compilation:

```
guile-snarf -o image-type.x image-type.c
```

This scans image-type.c for SCM_DEFINE declarations, and writes to image-type.x the output:

```
scm_c_define_gsubr ("clear-image", 1, 0, 0, (SCM (*)() ) clear_image);
```

When compiled normally, SCM_DEFINE is a macro which expands to the function header for clear_image.

Note that the output file name matches the #include from the input file. Also, you still need to provide all the same information you would if you were using scm_c_define_gsubr yourself, but you can place the information near the function definition itself, so it is less likely to become incorrect or out-of-date.

If you have many files that guile-snarf must process, you should consider using a fragment like the following in your Makefile:

```
snarfcppopts = $(DEFS) $(INCLUDES) $(CPPFLAGS) $(CFLAGS)
.SUFFIXES: .x
.c.x:
guile-snarf -o $@ $< $(snarfcppopts)
```

This tells make to run guile-snarf to produce each needed .x file from the corresponding .c file.

The program guile-snarf passes its command-line arguments directly to the C preprocessor, which it uses to extract the information it needs from the source code. This means you can pass normal compilation flags to guile-snarf to define preprocessor symbols, add header file directories, and so on.

### 5.7 An Overview of Guile Programming

Guile is designed as an extension language interpreter that is straightforward to integrate with applications written in C (and C++). The big win here for the application developer is that Guile integration, as the Guile web page says, “lowers your project’s hacktivation energy.” Lowering the hacktivation energy means that you, as the application developer, and your users, reap the benefits that flow from being able to extend the application in a high level extension language rather than in plain old C.

In abstract terms, it’s difficult to explain what this really means and what the integration process involves, so instead let’s begin by jumping straight into an example of how you might integrate Guile into an existing program, and what you could expect to gain by so doing. With that example under our belts, we’ll then return to a more general analysis of the arguments involved and the range of programming options available.

#### 5.7.1 How One Might Extend Dia Using Guile

Dia is a free software program for drawing schematic diagrams like flow charts and floor plans (http://www.gnome.org/projects/dia/). This section conducts the thought experiment of adding Guile to Dia. In so doing, it aims to illustrate several of the steps and considerations involved in adding Guile to applications in general.
5.7.1.1 Deciding Why You Want to Add Guile

First off, you should understand why you want to add Guile to Dia at all, and that means forming a picture of what Dia does and how it does it. So, what are the constituents of the Dia application?

- Most importantly, the application domain objects — in other words, the concepts that differentiate Dia from another application such as a word processor or spreadsheet: shapes, templates, connectors, pages, plus the properties of all these things.
- The code that manages the graphical face of the application, including the layout and display of the objects above.
- The code that handles input events, which indicate that the application user is wanting to do something.

(In other words, a textbook example of the model - view - controller paradigm.)

Next question: how will Dia benefit once the Guile integration is complete? Several (positive!) answers are possible here, and the choice is obviously up to the application developers. Still, one answer is that the main benefit will be the ability to manipulate Dia’s application domain objects from Scheme.

Suppose that Dia made a set of procedures available in Scheme, representing the most basic operations on objects such as shapes, connectors, and so on. Using Scheme, the application user could then write code that builds upon these basic operations to create more complex procedures. For example, given basic procedures to enumerate the objects on a page, to determine whether an object is a square, and to change the fill pattern of a single shape, the user can write a Scheme procedure to change the fill pattern of all squares on the current page:

```scheme
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
        (change-fill-pattern shape new-pattern))))
)
```

5.7.1.2 Four Steps Required to Add Guile

Assuming this objective, four steps are needed to achieve it.

First, you need a way of representing your application-specific objects — such as `shape` in the previous example — when they are passed into the Scheme world. Unless your objects are so simple that they map naturally into builtin Scheme data types like numbers and strings, you will probably want to use Guile’s foreign object interface to create a new Scheme data type for your objects.

Second, you need to write code for the basic operations like `for-each-shape` and `square?` such that they access and manipulate your existing data structures correctly, and then make these operations available as primitives on the Scheme level.

Third, you need to provide some mechanism within the Dia application that a user can hook into to cause arbitrary Scheme code to be evaluated.

Finally, you need to restructure your top-level application C code a little so that it initializes the Guile interpreter correctly and declares your foreign objects and primitives to the Scheme world.
The following subsections expand on these four points in turn.

### 5.7.1.3 How to Represent Dia Data in Scheme

For all but the most trivial applications, you will probably want to allow some representation of your domain objects to exist on the Scheme level. This is where foreign objects come in, and with them issues of lifetime management and garbage collection.

To get more concrete about this, let’s look again at the example we gave earlier of how application users can use Guile to build higher-level functions from the primitives that Dia itself provides.

```scheme
(define (change-squares’-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
         (change-fill-pattern shape new-pattern))))
```

Consider what is stored here in the variable `shape`. For each shape on the current page, the `for-each-shape` primitive calls `(lambda (shape) ...)` with an argument representing that shape. Question is: how is that argument represented on the Scheme level? The issues are as follows.

- Whatever the representation, it has to be decodable again by the C code for the `square?` and `change-fill-pattern` primitives. In other words, a primitive like `square?` has somehow to be able to turn the value that it receives back into something that points to the underlying C structure describing a shape.

- The representation must also cope with Scheme code holding on to the value for later use. What happens if the Scheme code stores `shape` in a global variable, but then that shape is deleted (in a way that the Scheme code is not aware of), and later on some other Scheme code uses that global variable again in a call to, say, `square?`?

- The lifetime and memory allocation of objects that exist only in the Scheme world is managed automatically by Guile’s garbage collector using one simple rule: when there are no remaining references to an object, the object is considered dead and so its memory is freed. But for objects that exist in both C and Scheme, the picture is more complicated; in the case of Dia, where the `shape` argument passes transiently in and out of the Scheme world, it would be quite wrong the delete the underlying C shape just because the Scheme code has finished evaluation. How do we avoid this happening?

One resolution of these issues is for the Scheme-level representation of a shape to be a new, Scheme-specific C structure wrapped up as a foreign object. The foreign object is what is passed into and out of Scheme code, and the Scheme-specific C structure inside the foreign object points to Dia’s underlying C structure so that the code for primitives like `square?` can get at it.

To cope with an underlying shape being deleted while Scheme code is still holding onto a Scheme shape value, the underlying C structure should have a new field that points to the Scheme-specific foreign object. When a shape is deleted, the relevant code chains through to the Scheme-specific structure and sets its pointer back to the underlying structure to NULL. Thus the foreign object value for the shape continues to exist, but any primitive
code that tries to use it will detect that the underlying shape has been deleted because the underlying structure pointer is NULL.

So, to summarize the steps involved in this resolution of the problem (and assuming that the underlying C structure for a shape is \texttt{struct dia\_shape}):

- Define a new Scheme-specific structure that points to the underlying C structure:
  \begin{verbatim}
  struct dia\_guile\_shape
  {
    struct dia\_shape * c\_shape; /* NULL => deleted */
  }
  \end{verbatim}

- Add a field to \texttt{struct dia\_shape} that points to its \texttt{struct dia\_guile\_shape} if it has one —
  \begin{verbatim}
  struct dia\_shape
  {
    ...
    struct dia\_guile\_shape * guile\_shape;
  }
  \end{verbatim}
  — so that C code can set \texttt{guile\_shape->c\_shape} to NULL when the underlying shape is deleted.

- Wrap \texttt{struct dia\_guile\_shape} as a foreign object type.

- Whenever you need to represent a C shape onto the Scheme level, create a foreign object instance for it, and pass that.

- In primitive code that receives a shape foreign object instance, check the \texttt{c\_shape} field when decoding it, to find out whether the underlying C shape is still there.

As far as memory management is concerned, the foreign object values and their Scheme-specific structures are under the control of the garbage collector, whereas the underlying C structures are explicitly managed in exactly the same way that Dia managed them before we thought of adding Guile.

When the garbage collector decides to free a shape foreign object value, it calls the finalizer function that was specified when defining the shape foreign object type. To maintain the correctness of the \texttt{guile\_shape} field in the underlying C structure, this function should chain through to the underlying C structure (if it still exists) and set its \texttt{guile\_shape} field to NULL.

For full documentation on defining and using foreign object types, see Section 5.5 [Defining New Foreign Object Types], page 74.

\textbf{5.7.1.4 Writing Guile Primitives for Dia}

Once the details of object representation are decided, writing the primitive function code that you need is usually straightforward.

A primitive is simply a C function whose arguments and return value are all of type \texttt{SCM}, and whose body does whatever you want it to do. As an example, here is a possible implementation of the \texttt{square?} primitive:

\begin{verbatim}
static SCM square_p (SCM shape)
{
  ...
}
\end{verbatim}
struct dia_guile_shape * guile_shape;

/* Check that arg is really a shape object. */
scm_assert_foreign_object_type (shape_type, shape);

/* Access Scheme-specific shape structure. */
guile_shape = scm_foreign_object_ref (shape, 0);

/* Find out if underlying shape exists and is a
square; return answer as a Scheme boolean. */
return scm_from_bool (guile_shape->c_shape &&
                      (guile_shape->c_shape->type == DIA_SQUARE));
}

Notice how easy it is to chain through from the SCM shape parameter that square_p
receives — which is a foreign object — to the Scheme-specific structure inside the foreign
object, and thence to the underlying C structure for the shape.

In this code, scm_assert_foreign_object_type, scm_foreign_object_ref, and scm_from_bool are from the standard Guile API. We assume that shape_type was given to
us when we made the shape foreign object type, using scm_make_foreign_object_type.
The call to scm_assert_foreign_object_type ensures that shape is indeed a shape. This
is needed to guard against Scheme code using the square? procedure incorrectly, as in
(square? "hello"); Scheme’s latent typing means that usage errors like this must be
cought at run time.

Having written the C code for your primitives, you need to make them available as
Scheme procedures by calling the scm_c_define_gsubr function. scm_c_define_gsubr
(see Section 6.9.2 [Primitive Procedures], page 249) takes arguments that specify the
Scheme-level name for the primitive and how many required, optional and rest arguments
it can accept. The square? primitive always requires exactly one argument, so the call to
make it available in Scheme reads like this:

scm_c_define_gsubr ("square?", 1, 0, 0, square_p);

For where to put this call, see the subsection after next on the structure of Guile-enabled
code (see Section 5.7.1.6 [Dia Structure], page 88).

5.7.1.5 Providing a Hook for the Evaluation of Scheme Code

To make the Guile integration useful, you have to design some kind of hook into your
application that application users can use to cause their Scheme code to be evaluated.

Technically, this is straightforward; you just have to decide on a mechanism that is
appropriate for your application. Think of Emacs, for example: when you type ESC ;, you
get a prompt where you can type in any Elisp code, which Emacs will then evaluate. Or,
again like Emacs, you could provide a mechanism (such as an init file) to allow Scheme
code to be associated with a particular key sequence, and evaluate the code when that key
sequence is entered.

In either case, once you have the Scheme code that you want to evaluate, as a null
terminated string, you can tell Guile to evaluate it by calling the scm_c_eval_string
function.
5.7.1.6 Top-level Structure of Guile-enabled Dia

Let’s assume that the pre-Guile Dia code looks structurally like this:

- `main ()`
  - do lots of initialization and setup stuff
  - enter Gtk main loop

When you add Guile to a program, one (rather technical) requirement is that Guile’s garbage collector needs to know where the bottom of the C stack is. The easiest way to ensure this is to use `scm_boot_guile` like this:

- `main ()`
  - do lots of initialization and setup stuff
  - `scm_boot_guile (argc, argv, inner_main, NULL)`
- `inner_main ()`
  - define all foreign object types
  - export primitives to Scheme using `scm_c_define_gsubr`
  - enter Gtk main loop

In other words, you move the guts of what was previously in your `main` function into a new function called `inner_main`, and then add a `scm_boot_guile` call, with `inner_main` as a parameter, to the end of `main`.

Assuming that you are using foreign objects and have written primitive code as described in the preceding subsections, you also need to insert calls to declare your new foreign objects and export the primitives to Scheme. These declarations must happen inside the dynamic scope of the `scm_boot_guile` call, but also before any code is run that could possibly use them — the beginning of `inner_main` is an ideal place for this.

5.7.1.7 Going Further with Dia and Guile

The steps described so far implement an initial Guile integration that already gives a lot of additional power to Dia application users. But there are further steps that you could take, and it’s interesting to consider a few of these.

In general, you could progressively move more of Dia’s source code from C into Scheme. This might make the code more maintainable and extensible, and it could open the door to new programming paradigms that are tricky to effect in C but straightforward in Scheme.

A specific example of this is that you could use the guile-gtk package, which provides Scheme-level procedures for most of the Gtk+ library, to move the code that lays out and displays Dia objects from C to Scheme.

As you follow this path, it naturally becomes less useful to maintain a distinction between Dia’s original non-Guile-related source code, and its later code implementing foreign objects and primitives for the Scheme world.

For example, suppose that the original source code had a `dia_change_fill_pattern` function:

```c
void dia_change_fill_pattern (struct dia_shape * shape, 
                              struct dia_pattern * pattern)
{
```
During initial Guile integration, you add a \texttt{change\_fill\_pattern} primitive for Scheme purposes, which accesses the underlying structures from its foreign object values and uses \texttt{dia\_change\_fill\_pattern} to do the real work:

\begin{verbatim}
SCM change_fill_pattern (SCM shape, SCM pattern)
{
    struct dia_shape * d_shape;
    struct dia_pattern * d_pattern;

    ...

    dia_change_fill_pattern (d_shape, d_pattern);

    return SCM_UNSPECIFIED;
}
\end{verbatim}

At this point, it makes sense to keep \texttt{dia\_change\_fill\_pattern} and \texttt{change\_fill\_pattern} separate, because \texttt{dia\_change\_fill\_pattern} can also be called without going through Scheme at all, say because the user clicks a button which causes a C-registered 

But, if the code for creating buttons and registering their callbacks is moved into Scheme (using guile-gtk), it may become true that \texttt{dia\_change\_fill\_pattern} can no longer be called other than through Scheme. In which case, it makes sense to abolish it and move its contents directly into \texttt{change\_fill\_pattern}, like this:

\begin{verbatim}
SCM change_fill_pattern (SCM shape, SCM pattern)
{
    struct dia_shape * d_shape;
    struct dia_pattern * d_pattern;

    ...

    /* real pattern change work */

    return SCM_UNSPECIFIED;
}
\end{verbatim}

So further Guile integration progressively reduces the amount of functional C code that you have to maintain over the long term.

A similar argument applies to data representation. In the discussion of foreign objects earlier, issues arose because of the different memory management and lifetime models that normally apply to data structures in C and in Scheme. However, with further Guile integration, you can resolve this issue in a more radical way by allowing all your data structures to be under the control of the garbage collector, and kept alive by references from the Scheme world. Instead of maintaining an array or linked list of shapes in C, you would instead maintain a list in Scheme.
Rather like the coalescing of `dia_change_fill_pattern` and `change_fill_pattern`, the practical upshot of such a change is that you would no longer have to keep the `dia_shape` and `dia_guile_shape` structures separate, and so wouldn’t need to worry about the pointers between them. Instead, you could change the foreign object definition to wrap the `dia_shape` structure directly, and send `dia_guile_shape` off to the scrap yard. Cut out the middle man!

Finally, we come to the holy grail of Guile’s free software / extension language approach. Once you have a Scheme representation for interesting Dia data types like shapes, and a handy bunch of primitives for manipulating them, it suddenly becomes clear that you have a bundle of functionality that could have far-ranging use beyond Dia itself. In other words, the data types and primitives could now become a library, and Dia becomes just one of the many possible applications using that library — albeit, at this early stage, a rather important one!

In this model, Guile becomes just the glue that binds everything together. Imagine an application that usefully combined functionality from Dia, Gnumeric and GnuCash — it’s tricky right now, because no such application yet exists; but it’ll happen some day . . .

### 5.7.2 Why Scheme is More Hackable Than C

Underlying Guile’s value proposition is the assumption that programming in a high level language, specifically Guile’s implementation of Scheme, is necessarily better in some way than programming in C. What do we mean by this claim, and how can we be so sure?

One class of advantages applies not only to Scheme, but more generally to any interpretable, high level, scripting language, such as Emacs Lisp, Python, Ruby, or TeX’s macro language. Common features of all such languages, when compared to C, are that:

- They lend themselves to rapid and experimental development cycles, owing usually to a combination of their interpretability and the integrated development environment in which they are used.
- They free developers from some of the low level bookkeeping tasks associated with C programming, notably memory management.
- They provide high level features such as container objects and exception handling that make common programming tasks easier.

In the case of Scheme, particular features that make programming easier — and more fun! — are its powerful mechanisms for abstracting parts of programs (closures — see Section 3.4 [About Closure], page 26) and for iteration (see Section 6.13.4 [while do], page 301).

The evidence in support of this argument is empirical: the huge amount of code that has been written in extension languages for applications that support this mechanism. Most notable are extensions written in Emacs Lisp for GNU Emacs, in TeX’s macro language for TeX, and in Script-Fu for the Gimp, but there is increasingly now a significant code eco-system for Guile-based applications as well, such as Lilypond and GnuCash. It is close to inconceivable that similar amounts of functionality could have been added to these applications just by writing new code in their base implementation languages.

### 5.7.3 Example: Using Guile for an Application Testbed

As an example of what this means in practice, imagine writing a testbed for an application that is tested by submitting various requests (via a C interface) and validating the
output received. Suppose further that the application keeps an idea of its current state, and that the “correct” output for a given request may depend on the current application state. A complete “white box”\(^4\) test plan for this application would aim to submit all possible requests in each distinguishable state, and validate the output for all request/state combinations.

To write all this test code in C would be very tedious. Suppose instead that the testbed code adds a single new C function, to submit an arbitrary request and return the response, and then uses Guile to export this function as a Scheme procedure. The rest of the testbed can then be written in Scheme, and so benefits from all the advantages of programming in Scheme that were described in the previous section.

(In this particular example, there is an additional benefit of writing most of the testbed in Scheme. A common problem for white box testing is that mistakes and mistaken assumptions in the application under test can easily be reproduced in the testbed code. It is more difficult to copy mistakes like this when the testbed is written in a different language from the application.)

### 5.7.4 A Choice of Programming Options

The preceding arguments and example point to a model of Guile programming that is applicable in many cases. According to this model, Guile programming involves a balance between C and Scheme programming, with the aim being to extract the greatest possible Scheme level benefit from the least amount of C level work.

The C level work required in this model usually consists of packaging and exporting functions and application objects such that they can be seen and manipulated on the Scheme level. To help with this, Guile’s C language interface includes utility features that aim to make this kind of integration very easy for the application developer. These features are documented later in this part of the manual: see REFFIXME.

This model, though, is really just one of a range of possible programming options. If all of the functionality that you need is available from Scheme, you could choose instead to write your whole application in Scheme (or one of the other high level languages that Guile supports through translation), and simply use Guile as an interpreter for Scheme. (In the future, we hope that Guile will also be able to compile Scheme code, so lessening the performance gap between C and Scheme code.) Or, at the other end of the C–Scheme scale, you could write the majority of your application in C, and only call out to Guile occasionally for specific actions such as reading a configuration file or executing a user-specified extension. The choices boil down to two basic questions:

- Which parts of the application do you write in C, and which in Scheme (or another high level translated language)?
- How do you design the interface between the C and Scheme parts of your application?

These are of course design questions, and the right design for any given application will always depend upon the particular requirements that you are trying to meet. In the context of Guile, however, there are some generally applicable considerations that can help you when designing your answers.

\(^4\) A white box test plan is one that incorporates knowledge of the internal design of the application under test.
5.7.4.1 What Functionality is Already Available?

Suppose, for the sake of argument, that you would prefer to write your whole application in Scheme. Then the API available to you consists of:

- standard Scheme
- plus the extensions to standard Scheme provided by Guile in its core distribution
- plus any additional functionality that you or others have packaged so that it can be loaded as a Guile Scheme module.

A module in the last category can either be a pure Scheme module — in other words a collection of utility procedures coded in Scheme — or a module that provides a Scheme interface to an extension library coded in C — in other words a nice package where someone else has done the work of wrapping up some useful C code for you. The set of available modules is growing quickly and already includes such useful examples as (gtk gtk), which makes Gtk+ drawing functions available in Scheme, and (database postgres), which provides SQL access to a Postgres database.

Given the growing collection of pre-existing modules, it is quite feasible that your application could be implemented by combining a selection of these modules together with new application code written in Scheme.

If this approach is not enough, because the functionality that your application needs is not already available in this form, and it is impossible to write the new functionality in Scheme, you will need to write some C code. If the required function is already available in C (e.g. in a library), all you need is a little glue to connect it to the world of Guile. If not, you need both to write the basic code and to plumb it into Guile.

In either case, two general considerations are important. Firstly, what is the interface by which the functionality is presented to the Scheme world? Does the interface consist only of function calls (for example, a simple drawing interface), or does it need to include objects of some kind that can be passed between C and Scheme and manipulated by both worlds. Secondly, how does the lifetime and memory management of objects in the C code relate to the garbage collection governed approach of Scheme objects? In the case where the basic C code is not already written, most of the difficulties of memory management can be avoided by using Guile’s C interface features from the start.

For the full documentation on writing C code for Guile and connecting existing C code to the Guile world, see REFFIXME.

5.7.4.2 Functional and Performance Constraints

5.7.4.3 Your Preferred Programming Style

5.7.4.4 What Controls Program Execution?

5.7.5 How About Application Users?

So far we have considered what Guile programming means for an application developer. But what if you are instead using an existing Guile-based application, and want to know what your options are for programming and extending this application?

The answer to this question varies from one application to another, because the options available depend inevitably on whether the application developer has provided any hooks
for you to hang your own code on and, if there are such hooks, what they allow you to do.\textsuperscript{5} For example... 

- If the application permits you to load and execute any Guile code, the world is your oyster. You can extend the application in any way that you choose.
- A more cautious application might allow you to load and execute Guile code, but only in a safe environment, where the interface available is restricted by the application from the standard Guile API.
- Or a really fearful application might not provide a hook to really execute user code at all, but just use Scheme syntax as a convenient way for users to specify application data or configuration options.

In the last two cases, what you can do is, by definition, restricted by the application, and you should refer to the application’s own manual to find out your options.

The most well known example of the first case is Emacs, with its extension language Emacs Lisp: as well as being a text editor, Emacs supports the loading and execution of arbitrary Emacs Lisp code. The result of such openness has been dramatic: Emacs now benefits from user-contributed Emacs Lisp libraries that extend the basic editing function to do everything from reading news to psychoanalysis and playing adventure games. The only limitation is that extensions are restricted to the functionality provided by Emacs’s built-in set of primitive operations. For example, you can interact and display data by manipulating the contents of an Emacs buffer, but you can’t pop-up and draw a window with a layout that is totally different to the Emacs standard.

This situation with a Guile application that supports the loading of arbitrary user code is similar, except perhaps even more so, because Guile also supports the loading of extension libraries written in C. This last point enables user code to add new primitive operations to Guile, and so to bypass the limitation present in Emacs Lisp.

At this point, the distinction between an application developer and an application user becomes rather blurred. Instead of seeing yourself as a user extending an application, you could equally well say that you are developing a new application of your own using some of the primitive functionality provided by the original application. As such, all the discussions of the preceding sections of this chapter are relevant to how you can proceed with developing your extension.

\textsuperscript{5} Of course, in the world of free software, you always have the freedom to modify the application’s source code to your own requirements. Here we are concerned with the extension options that the application has provided for without your needing to modify its source code.
5.8 Autoconf Support

Autoconf, a part of the GNU build system, makes it easy for users to build your package. This section documents Guile’s Autoconf support.

5.8.1 Autoconf Background

As explained in the GNU Autoconf Manual, any package needs configuration at build-time (see Section “Introduction” in The GNU Autoconf Manual). If your package uses Guile (or uses a package that in turn uses Guile), you probably need to know what specific Guile features are available and details about them.

The way to do this is to write feature tests and arrange for their execution by the configure script, typically by adding the tests to configure.ac, and running autoconf to create configure. Users of your package then run configure in the normal way.

Macros are a way to make common feature tests easy to express. Autoconf provides a wide range of macros (see Section “Existing Tests” in The GNU Autoconf Manual), and Guile installation provides Guile-specific tests in the areas of: program detection, compilation flags reporting, and Scheme module checks.

5.8.2 Autoconf Macros

As mentioned earlier in this chapter, Guile supports parallel installation, and uses pkg-config to let the user choose which version of Guile they are interested in. pkg-config has its own set of Autoconf macros that are probably installed on most every development system. The most useful of these macros is PKG_CHECK_MODULES.

\[
\text{PKG}_\text{CHECK}_\text{MODULES}([\text{GUILE}], [\text{guile-2.2}])
\]

This example looks for Guile and sets the GUILE_CFLAGS and GUILE_LIBS variables accordingly, or prints an error and exits if Guile was not found.

Guile comes with additional Autoconf macros providing more information, installed as prefix/share/aclocal/guile.m4. Their names all begin with GUILE_.

**GUILE_PKG [VERSIONS]**

This macro runs the pkg-config tool to find development files for an available version of Guile.

By default, this macro will search for the latest stable version of Guile (e.g. 2.2), falling back to the previous stable version (e.g. 2.0) if it is available. If no guile-VERSION.pc file is found, an error is signalled. The found version is stored in GUILE_EFFECTIVE_VERSION.

If GUILE_PROGS was already invoked, this macro ensures that the development files have the same effective version as the Guile program.

GUILE_EFFECTIVE_VERSION is marked for substitution, as by AC_SUBST.

**GUILE_FLAGS**

This macro runs the pkg-config tool to find out how to compile and link programs against Guile. It sets four variables: GUILE_CFLAGS, GUILE_LDFLAGS, GUILE_LIBS, and GUILE_LTLIBS.

GUILE_CFLAGS: flags to pass to a C or C++ compiler to build code that uses Guile header files. This is almost always just one or more -I flags.
**GUILE_LDFLAGS**: flags to pass to the compiler to link a program against Guile. This includes `-lguile-VERSION` for the Guile library itself, and may also include one or more `-L` flag to tell the compiler where to find the libraries. But it does not include flags that influence the program’s runtime search path for libraries, and will therefore lead to a program that fails to start, unless all necessary libraries are installed in a standard location such as `/usr/lib`.

**GUILE_LIBS** and **GUILE_LTLIBS**: flags to pass to the compiler or to libtool, respectively, to link a program against Guile. It includes flags that augment the program’s runtime search path for libraries, so that shared libraries will be found at the location where they were during linking, even in non-standard locations. **GUILE_LIBS** is to be used when linking the program directly with the compiler, whereas **GUILE_LTLIBS** is to be used when linking the program is done through libtool.

The variables are marked for substitution, as by `AC_SUBST`.

**GUILE_SITE_DIR**

This looks for Guile's "site" directories. The variable **GUILE_SITE** will be set to Guile’s "site" directory for Scheme source files (usually something like `PREFIX/share/guile/site`). **GUILE_SITE_CCACHE** will be set to the directory for compiled Scheme files also known as `.go` files (usually something like `PREFIX/lib/guile/GUILE_EFFECTIVE_VERSION/site-ccache`). **GUILE_EXTENSION** will be set to the directory for compiled C extensions (usually something like `PREFIX/lib/guile/GUILE_EFFECTIVE_VERSION/extensions`). The latter two are set to blank if the particular version of Guile does not support them. Note that this macro will run the macros **GUILE_PKG** and **GUILE_PROGS** if they have not already been run.

The variables are marked for substitution, as by `AC_SUBST`.

**GUILE_PROGS [VERSION]**

This macro looks for programs `guile` and `guild`, setting variables **GUILE** and **GUILD** to their paths, respectively. The macro will attempt to find `guile` with the suffix of `-X.Y`, followed by looking for it with the suffix `X.Y`, and then fall back to looking for `guile` with no suffix. If `guile` is still not found, signal an error. The suffix, if any, that was required to find `guile` will be used for `guild` as well.

By default, this macro will search for the latest stable version of Guile (e.g. 2.2). x.y or x.y.z versions can be specified. If an older version is found, the macro will signal an error.

The effective version of the found `guile` is set to **GUILE_EFFECTIVE_VERSION**. This macro ensures that the effective version is compatible with the result of a previous invocation of **GUILE_FLAGS**, if any.

As a legacy interface, it also looks for `guile-config` and `guile-tools`, setting **GUILE_CONFIG** and **GUILE_TOOLS**.

The variables are marked for substitution, as by `AC_SUBST`.

**GUILE_CHECK_RETVAL**

`var check`

`var` is a shell variable name to be set to the return value. `check` is a Guile Scheme expression, evaluated with "\$GUILE -c", and returning either 0 or non-#f to indicate
the check passed. Non-0 number or #f indicates failure. Avoid using the character "#" since that confuses autoconf.

**GUILE_MODULE_CHECK** \texttt{var module featuretest description} \ [Autoconf Macro]
\texttt{var} is a shell variable name to be set to "yes" or "no". \texttt{module} is a list of symbols, like: (ice-9 common-list). \texttt{featuretest} is an expression acceptable to \texttt{GUILE.CHECK}, q.v. \texttt{description} is a present-tense verb phrase (passed to \texttt{AC_MSG_CHECKING}).

**GUILE_MODULE_AVAILABLE** \texttt{var module} \ [Autoconf Macro]
\texttt{var} is a shell variable name to be set to "yes" or "no". \texttt{module} is a list of symbols, like: (ice-9 common-list).

**GUILE_MODULE_REQUIRED** \texttt{symlist} \ [Autoconf Macro]
\texttt{symlist} is a list of symbols, WITHOUT surrounding parens, like: ice-9 common-list.

**GUILE_MODULE_EXPORTS** \texttt{var module modvar} \ [Autoconf Macro]
\texttt{var} is a shell variable to be set to "yes" or "no". \texttt{module} is a list of symbols, like: (ice-9 common-list). \texttt{modvar} is the Guile Scheme variable to check.

**GUILE_MODULE_REQUIRED_EXPORT** \texttt{module modvar} \ [Autoconf Macro]
\texttt{module} is a list of symbols, like: (ice-9 common-list). \texttt{modvar} is the Guile Scheme variable to check.

### 5.8.3 Using Autoconf Macros

Using the autoconf macros is straightforward: Add the macro "calls" (actually instantiations) to \texttt{configure.ac}, run \texttt{aclocal}, and finally, run \texttt{autoconf}. If your system doesn’t have guile.m4 installed, place the desired macro definitions (\texttt{AC_DEFUN} forms) in \texttt{acinclude.m4}, and \texttt{aclocal} will do the right thing.

Some of the macros can be used inside normal shell constructs: \texttt{if foo ; then GUILE_BAZ ; fi}, but this is not guaranteed. It’s probably a good idea to instantiate macros at top-level.

We now include two examples, one simple and one complicated.

The first example is for a package that uses libguile, and thus needs to know how to compile and link against it. So we use \texttt{PKG_CHECK_MODULES} to set the vars \texttt{GUILE_CFLAGS} and \texttt{GUILE_LIBS}, which are automatically substituted in the Makefile.

In \texttt{configure.ac}:

```bash
PKG_CHECK_MODULES([GUILE], [guile-2.2])
```

In \texttt{Makefile.in}:

```bash
GUILE_CFLAGS = @GUILE_CFLAGS@
GUILE_LIBS = @GUILE_LIBS@

myprog.o: myprog.c
    $(CC) -o $ $(GUILE_CFLAGS) $<
myprog: myprog.o
```
The second example is for a package of Guile Scheme modules that uses an external program and other Guile Scheme modules (some might call this a "pure scheme" package). So we use the GUILE_SITE_DIR macro, a regular AC_PATH_PROG macro, and the GUILE_MODULE_AVAILABLE macro.

In configure.ac:

GUILE_SITE_DIR

probably_wont_work=""

# pgtype pgtable
GUILE_MODULE_AVAILABLE(have_guile_pg, (database postgres))
test $have_guile_pg = no &&
    probably_wont_work="(my pgtype) (my pgtable) $probably_wont_work"

# gpgutils
AC_PATH_PROG(GNUPG,gpg)
test x"$GNUPG" = x &&
    probably_wont_work="(my gpgutils) $probably_wont_work"

if test ! "$probably_wont_work" = "" ; then
    p="""" ***"
    echo
    echo "$p"
    echo "$p NOTE:"
    echo "$p The following modules probably won't work:"
    echo "$p $probably_wont_work"
    echo "$p They can be installed anyway, and will work if their"
    echo "$p dependencies are installed later. Please see README."
    echo "$p"
    echo
fi

In Makefile.in:

instdir = @GUILE_SITE@/my

install:
    $(INSTALL) my/*.scm $(instdir)
6 API Reference

Guile provides an application programming interface (API) to developers in two core languages: Scheme and C. This part of the manual contains reference documentation for all of the functionality that is available through both Scheme and C interfaces.

6.1 Overview of the Guile API

Guile’s application programming interface (API) makes functionality available that an application developer can use in either C or Scheme programming. The interface consists of elements that may be macros, functions or variables in C, and procedures, variables, syntax or other types of object in Scheme.

Many elements are available to both Scheme and C, in a form that is appropriate. For example, the assq Scheme procedure is also available as scm_assq to C code. These elements are documented only once, addressing both the Scheme and C aspects of them.

The Scheme name of an element is related to its C name in a regular way. Also, a C function takes its parameters in a systematic way.

Normally, the name of a C function can be derived given its Scheme name, using some simple textual transformations:

- Replace - (hyphen) with _ (underscore).
- Replace ? (question mark) with _p.
- Replace ! (exclamation point) with _x.
- Replace internal -> with _to_.
- Replace <= (less than or equal) with _leq.
- Replace >= (greater than or equal) with _geq.
- Replace < (less than) with _less.
- Replace > (greater than) with _gr.
- Prefix with scm_.

A C function always takes a fixed number of arguments of type SCM, even when the corresponding Scheme function takes a variable number.

For some Scheme functions, some last arguments are optional; the corresponding C function must always be invoked with all optional arguments specified. To get the effect as if an argument has not been specified, pass SCM_UNDEFINED as its value. You can not do this for an argument in the middle; when one argument is SCM_UNDEFINED all the ones following it must be SCM_UNDEFINED as well.

Some Scheme functions take an arbitrary number of rest arguments; the corresponding C function must be invoked with a list of all these arguments. This list is always the last argument of the C function.

These two variants can also be combined.

The type of the return value of a C function that corresponds to a Scheme function is always SCM. In the descriptions below, types are therefore often omitted but for the return value and for the arguments.
6.2 Deprecation

From time to time functions and other features of Guile become obsolete. Guile’s deprecation is a mechanism that can help you cope with this.

When you use a feature that is deprecated, you will likely get a warning message at runtime. Also, if you have a new enough toolchain, using a deprecated function from `libguile` will cause a link-time warning.

The primary source for information about just what interfaces are deprecated in a given release is the file `NEWS`. That file also documents what you should use instead of the obsoleted things.

The file `README` contains instructions on how to control the inclusion or removal of the deprecated features from the public API of Guile, and how to control the deprecation warning messages.

The idea behind this mechanism is that normally all deprecated interfaces are available, but you get feedback when compiling and running code that uses them, so that you can migrate to the newer APIs at your leisure.

6.3 The SCM Type

Guile represents all Scheme values with the single C type `SCM`. For an introduction to this topic, See Section 5.4.1 [Dynamic Types], page 65.

```
SCM
```

`SCM` is the user level abstract C type that is used to represent all of Guile’s Scheme objects, no matter what the Scheme object type is. No C operation except assignment is guaranteed to work with variables of type `SCM`, so you should only use macros and functions to work with `SCM` values. Values are converted between C data types and the `SCM` type with utility functions and macros.

```
scm_t_bits
```

`scm_t_bits` is an unsigned integral data type that is guaranteed to be large enough to hold all information that is required to represent any Scheme object. While this data type is mostly used to implement Guile’s internals, the use of this type is also necessary to write certain kinds of extensions to Guile.

```
scm_t_signed_bits
```

This is a signed integral type of the same size as `scm_t_bits`.

```
scm_t_bits SCM_UNPACK (SCM x)
```

Transforms the `SCM` value `x` into its representation as an integral type. Only after applying `SCM_UNPACK` it is possible to access the bits and contents of the `SCM` value.

```
SCM SCM_PACK (scm_t_bits x)
```

Takes a valid integral representation of a Scheme object and transforms it into its representation as a `SCM` value.
6.4 Initializing Guile

Each thread that wants to use functions from the Guile API needs to put itself into guile mode with either \texttt{scm\_with\_guile} or \texttt{scm\_init\_guile}. The global state of Guile is initialized automatically when the first thread enters guile mode.

When a thread wants to block outside of a Guile API function, it should leave guile mode temporarily with \texttt{scm\_without\_guile}, See Section 6.22.6 [Blocking], page 446.

Threads that are created by \texttt{call\-with\-new\-thread} or \texttt{scm\_spawn\_thread} start out in guile mode so you don’t need to initialize them.

\begin{verbatim}
void * scm_with_guile (void *(*func)(void *), void *data)              [C Function]
Call func, passing it data and return what func returns. While func is running, the current thread is in guile mode and can thus use the Guile API.

When scm_with_guile is called from guile mode, the thread remains in guile mode when scm_with_guile returns.

Otherwise, it puts the current thread into guile mode and, if needed, gives it a Scheme representation that is contained in the list returned by all-threads, for example. This Scheme representation is not removed when scm_with_guile returns so that a given thread is always represented by the same Scheme value during its lifetime, if at all.

When this is the first thread that enters guile mode, the global state of Guile is initialized before calling func.

The function func is called via scm_with_continuation_barrier; thus, scm_with_guile returns exactly once.

When scm_with_guile returns, the thread is no longer in guile mode (except when scm_with_guile was called from guile mode, see above). Thus, only func can store SCM variables on the stack and be sure that they are protected from the garbage collector. See scm_init_guile for another approach at initializing Guile that does not have this restriction.

It is OK to call scm_with_guile while a thread has temporarily left guile mode via scm_without_guile. It will then simply temporarily enter guile mode again.

void scm_init_guile ()                           [C Function]
Arrange things so that all of the code in the current thread executes as if from within a call to scm_with_guile. That is, all functions called by the current thread can assume that SCM values on their stack frames are protected from the garbage collector (except when the thread has explicitly left guile mode, of course).

When scm_init_guile is called from a thread that already has been in guile mode once, nothing happens. This behavior matters when you call scm_init_guile while the thread has only temporarily left guile mode: in that case the thread will not be in guile mode after scm_init_guile returns. Thus, you should not use scm_init_guile in such a scenario.

When an uncaught throw happens in a thread that has been put into guile mode via scm_init_guile, a short message is printed to the current error port and the thread is exited via scm_pthread_exit (NULL). No restrictions are placed on continuations.
\end{verbatim}
The function `scm_init_guile` might not be available on all platforms since it requires some stack-bounds-finding magic that might not have been ported to all platforms that Guile runs on. Thus, if you can, it is better to use `scm_with_guile` or its variation `scm_boot_guile` instead of this function.

```c
void scm_boot_guile (int argc, char **argv, void (*main_func) (void *, int argc, char **argv), void *data)
```

Enter guile mode as with `scm_with_guile` and call `main_func`, passing it `data`, `argc`, and `argv` as indicated. When `main_func` returns, `scm_boot_guile` calls `exit (0)`. `scm_boot_guile` never returns. If you want some other exit value, have `main_func` call `exit` itself. If you don’t want to exit at all, use `scm_with_guile` instead of `scm_boot_guile`.

The function `scm_boot_guile` arranges for the Scheme command-line function to return the strings given by `argc` and `argv`. If `main_func` modifies `argc` or `argv`, it should call `scm_set_program_arguments` with the final list, so Scheme code will know which arguments have been processed (see Section 7.2.6 [Runtime Environment], page 512).

```c
void scm_shell (int argc, char **argv)
```

Process command-line arguments in the manner of the `guile` executable. This includes loading the normal Guile initialization files, interacting with the user or running any scripts or expressions specified by `-s` or `-e` options, and then exiting. See Section 4.2 [Invoking Guile], page 35, for more details.

Since this function does not return, you must do all application-specific initialization before calling this function.

### 6.5 Snarfing Macros

The following macros do two different things: when compiled normally, they expand in one way; when processed during snarfing, they cause the `guile-snarf` program to pick up some initialization code, See Section 5.6 [Function Snarfing], page 81.

The descriptions below use the term ‘normally’ to refer to the case when the code is compiled normally, and ‘while snarfing’ when the code is processed by `guile-snarf`.

**SCM_SNARF_INIT** *(code)*

Normally, `SCM_SNARF_INIT` expands to nothing; while snarfing, it causes `code` to be included in the initialization action file, followed by a semicolon.

This is the fundamental macro for snarfing initialization actions. The more specialized macros below use it internally.

**SCM_DEFINE** *(c_name, scheme_name, req, opt, var, arglist, docstring)*

Normally, this macro expands into

```c
static const char s_c_name[] = scheme_name;
SCM
   c_name arglist
```

While snarfing, it causes

```c
scm_c_define_gsubr (s_c_name, req, opt, var,
   c_name);
```
to be added to the initialization actions. Thus, you can use it to declare a C function
named c_name that will be made available to Scheme with the name scheme_name.
Note that the arglist argument must have parentheses around it.

**SCM_SYMBOL (c_name, scheme_name)**

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_from_locale_symbol (scheme_name));
```

Thus, you can use them declare a static or global variable of type SCM that will be
initialized to the symbol named scheme_name.

**SCM_KEYWORD (c_name, scheme_name)**

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_c_make_keyword (scheme_name));
```

Thus, you can use them declare a static or global variable of type SCM that will be
initialized to the keyword named scheme_name.

**SCM_VARIABLE (c_name, scheme_name)**

These macros are equivalent to **SCM_VARIABLE_INIT** and **SCM_GLOBAL_VARIABLE_INIT**, respectively, with a value of SCM_BOOL_F.

**SCM_VARIABLE_INIT (c_name, scheme_name, value)**

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_c_define (scheme_name, value));
```

Thus, you can use them declare a static or global C variable of type SCM that will be
initialized to the object representing the Scheme variable named scheme_name in
the current module. The variable will be defined when it doesn’t already exist. It is
always set to value.
6.6 Data Types

Guile’s data types form a powerful built-in library of representations and functionality that you can apply to your problem domain. This chapter surveys the data types built-in to Guile, from the simple to the complex.

6.6.1 Booleans

The two boolean values are `#t` for true and `#f` for false. They can also be written as `#true` and `#false`, as per R7RS.

Boolean values are returned by predicate procedures, such as the general equality predicates `eq?`, `eqv?` and `equal?` (see Section 6.11.1 [Equality], page 283) and numerical and string comparison operators like `string=` (see Section 6.6.5.7 [String Comparison], page 148) and `<=` (see Section 6.6.2.8 [Comparison], page 117).

```
(<= 3 8)  ⇒  #t
(<= 3 -3) ⇒  #f

(equal? "house" "houses") ⇒ #f

(eq? #f #f) ⇒ #t
```

In test condition contexts like `if` and `cond` (see Section 6.13.2 [Conditionals], page 299), where a group of subexpressions will be evaluated only if a condition expression evaluates to “true”, “true” means any value at all except `#f`.

```
(if #t "yes" "no")  ⇒  "yes"
(if 0 "yes" "no")  ⇒  "yes"
(if #f "yes" "no")  ⇒  "no"
```

A result of this asymmetry is that typical Scheme source code more often uses `#f` explicitly than `#t`: `#f` is necessary to represent an `if` or `cond` false value, whereas `#t` is not necessary to represent an `if` or `cond` true value.

It is important to note that `#f` is not equivalent to any other Scheme value. In particular, `#f` is not the same as the number 0 (like in C and C++), and not the same as the “empty list” (like in some Lisp dialects).

In C, the two Scheme boolean values are available as the two constants `SCM_BOOL_T` for `#t` and `SCM_BOOL_F` for `#f`. Care must be taken with the false value `SCM_BOOL_F`: it is not false when used in C conditionals. In order to test for it, use `scm_is_false` or `scm_is_true`. 
Chapter 6: API Reference

not x

scm_not (x)  
Return #t if x is #f, else return #f.

boolean? obj

scm_boolean_p (obj)
Return #t if obj is either #t or #f, else return #f.

SCM SCM_BOOL_T
The SCM representation of the Scheme object #t.

SCM SCM_BOOL_F
The SCM representation of the Scheme object #f.

int scm_is_true (SCM obj)
Return 0 if obj is #f, else return 1.

int scm_is_false (SCM obj)
Return 1 if obj is #f, else return 0.

int scm_is_bool (SCM obj)
Return 1 if obj is either #t or #f, else return 0.

SCM scm_from_bool (int val)
Return #f if val is 0, else return #t.

int scm_to_bool (SCM val)
Return 1 if val is SCM_BOOL_T, return 0 when val is SCM_BOOL_F, else signal a 'wrong type' error.
You should probably use scm_is_true instead of this function when you just want to test a SCM value for trueness.

6.6.2 Numerical data types

Guile supports a rich “tower” of numerical types — integer, rational, real and complex — and provides an extensive set of mathematical and scientific functions for operating on numerical data. This section of the manual documents those types and functions.

You may also find it illuminating to read R5RS’s presentation of numbers in Scheme, which is particularly clear and accessible: see Section “Numbers” in R5RS.

6.6.2.1 Scheme’s Numerical “Tower”

Scheme’s numerical “tower” consists of the following categories of numbers:

- **integers**: Whole numbers, positive or negative; e.g. -5, 0, 18.
- **rationals**: The set of numbers that can be expressed as \( \frac{p}{q} \) where \( p \) and \( q \) are integers; e.g. 9/16 works, but pi (an irrational number) doesn’t. These include integers \( \frac{n}{1} \).
- **real numbers**: The set of numbers that describes all possible positions along a one-dimensional line. This includes rationals as well as irrational numbers.
complex numbers

The set of numbers that describes all possible positions in a two dimensional space. This includes real as well as imaginary numbers \((a + bi, \text{ where } a \text{ is the real part, } b \text{ is the imaginary part, and } i \text{ is the square root of } -1)\).

It is called a tower because each category “sits on” the one that follows it, in the sense that every integer is also a rational, every rational is also real, and every real number is also a complex number (but with zero imaginary part).

In addition to the classification into integers, rationals, reals and complex numbers, Scheme also distinguishes between whether a number is represented exactly or not. For example, the result of \(2 \sin(\pi/4)\) is exactly \(\sqrt{2}\), but Guile can represent neither \(\pi/4\) nor \(\sqrt{2}\) exactly. Instead, it stores an inexact approximation, using the C type `double`.

Guile can represent exact rationals of any magnitude, inexact rationals that fit into a C `double`, and inexact complex numbers with `double` real and imaginary parts.

The `number?` predicate may be applied to any Scheme value to discover whether the value is any of the supported numerical types.

```scheme
(define pi 3.141592654)
(number? pi)  => #t
```

For example:

```scheme
(number? 3)  => #t
(number? "hello there!")  => #f
(define pi 3.141592654)
(number? pi)  => #t
```

```c
int scm_is_number (SCM obj)
This is equivalent to scm_is_true (scm_number_p (obj)).
```

The next few subsections document each of Guile’s numerical data types in detail.

6.6.2.2 Integers

Integers are whole numbers, that is numbers with no fractional part, such as 2, 83, and -3789.

Integers in Guile can be arbitrarily big, as shown by the following example.

```scheme
(define (factorial n)
  (let loop ((n n) (product 1))
    (if (= n 0)
      product
      (loop (- n 1) (* product n)))))
```
Readers whose background is in programming languages where integers are limited by the need to fit into just 4 or 8 bytes of memory may find this surprising, or suspect that Guile’s representation of integers is inefficient. In fact, Guile achieves a near optimal balance of convenience and efficiency by using the host computer’s native representation of integers where possible, and a more general representation where the required number does not fit in the native form. Conversion between these two representations is automatic and completely invisible to the Scheme level programmer.

C has a host of different integer types, and Guile offers a host of functions to convert between them and the SCM representation. For example, a C int can be handled with scm_to_int and scm_from_int. Guile also defines a few C integer types of its own, to help with differences between systems.

C integer types that are not covered can be handled with the generic scm_to_signed_integer and scm_from_signed_integer for signed types, or with scm_to_unsigned_integer and scm_from_unsigned_integer for unsigned types.

Scheme integers can be exact and inexact. For example, a number written as 3.0 with an explicit decimal-point is inexact, but it is also an integer. The functions integer? and scm_is_integer report true for such a number, but the functions exact-integer?, scm_is_exact_integer, scm_is_signed_integer, and scm_is_unsigned_integer only allow exact integers and thus report false. Likewise, the conversion functions like scm_to_signed_integer only accept exact integers.

The motivation for this behavior is that the inexactness of a number should not be lost silently. If you want to allow inexact integers, you can explicitly insert a call to inexact->exact or to its C equivalent scm_inexact_to_exact. (Only inexact integers will be converted by this call into exact integers; inexact non-integers will become exact fractions.)

(integer? 487) ⇒ #t

(integer? 3.0) ⇒ #t

(integer? -3.4) ⇒ #f
(integer? +inf.0)  
⇒ #f

int scm_is_integer (SCM x)  
This is equivalent to scm_is_true (scm_integer_p (x)).

exact-integer? x  
scm_exact_integer_p (x)  
Return #t if x is an exact integer number, else return #f.

(exact-integer? 37)  
⇒ #t

(exact-integer? 3.0)  
⇒ #f

int scm_is_exact_integer (SCM x)  
This is equivalent to scm_is_true (scm_exact_integer_p (x)).

scm_t_int8  
scm_t_uint8  
scm_t_int16  
scm_t_uint16  
scm_t_int32  
scm_t_uint32  
scm_t_int64  
scm_t_uint64  
scm_t_intmax  
scm_t_uintmax  
The C types are equivalent to the corresponding ISO C types but are defined on all platforms, with the exception of scm_t_int64 and scm_t_uint64, which are only defined when a 64-bit type is available. For example, scm_t_int8 is equivalent to int8_t.

You can regard these definitions as a stop-gap measure until all platforms provide these types. If you know that all the platforms that you are interested in already provide these types, it is better to use them directly instead of the types provided by Guile.

int scm_is_signed_integer (SCM x, scm_t_intmax min,  
scm_t_intmax max)  
int scm_is_unsigned_integer (SCM x, scm_t_uintmax min,  
scm_t_uintmax max)  
Return 1 when x represents an exact integer that is between min and max, inclusive.

These functions can be used to check whether a SCM value will fit into a given range, such as the range of a given C integer type. If you just want to convert a SCM value to a given C integer type, use one of the conversion functions directly.
scm_t_intmax scm_to_signed_integer (SCM x, scm_t_intmax min, scm_t_intmax max)  
[C Function]

scm_t_uintmax scm_to_unsigned_integer (SCM x, 
scm_t_uintmax min, scm_t_uintmax max)  
[C Function]

When x represents an exact integer that is between min and max inclusive, return 
that integer. Else signal an error, either a ‘wrong-type’ error when x is not an exact 
integer, or an ‘out-of-range’ error when it doesn’t fit the given range.

SCM scm_from_signed_integer (scm_t_intmax x)  
[C Function]

SCM scm_from_unsigned_integer (scm_t_uintmax x)  
[C Function]

Return the SCM value that represents the integer x. This function will always succeed 
and will always return an exact number.

char scm_to_char (SCM x)  
[C Function]

signed char scm_to_schar (SCM x)  
[C Function]

unsigned char scm_to_uchar (SCM x)  
[C Function]

short scm_to_short (SCM x)  
[C Function]

unsigned short scm_to_ushort (SCM x)  
[C Function]

int scm_to_int (SCM x)  
[C Function]

unsigned int scm_to_uint (SCM x)  
[C Function]

long scm_to_long (SCM x)  
[C Function]

unsigned long scm_to_ulong (SCM x)  
[C Function]

long long scm_to_long_long (SCM x)  
[C Function]

unsigned long long scm_to_ulonglong_long (SCM x)  
[C Function]

size_t scm_to_size_t (SCM x)  
[C Function]

ssize_t scm_to_ssize_t (SCM x)  
[C Function]

scm_t_uintptr scm_to_uintptr_t (SCM x)  
[C Function]

scm_t_ptrdiff scm_to_ptrdiff_t (SCM x)  
[C Function]

scm_t_int8 scm_to_int8 (SCM x)  
[C Function]

scm_t_uint8 scm_to_uint8 (SCM x)  
[C Function]

scm_t_int16 scm_to_int16 (SCM x)  
[C Function]

scm_t_uint16 scm_to_uint16 (SCM x)  
[C Function]

scm_t_int32 scm_to_int32 (SCM x)  
[C Function]

scm_t_uint32 scm_to_uint32 (SCM x)  
[C Function]

scm_t_int64 scm_to_int64 (SCM x)  
[C Function]

scm_t_uint64 scm_to_uint64 (SCM x)  
[C Function]

scm_t_intmax scm_to_intmax (SCM x)  
[C Function]

scm_t_uintmax scm_to_uintmax (SCM x)  
[C Function]

scm_t_intptr scm_to_intptr_t (SCM x)  
[C Function]

scm_t_uintptr scm_to_uintptr_t (SCM x)  
[C Function]

When x represents an exact integer that fits into the indicated C type, return that 
integer. Else signal an error, either a ‘wrong-type’ error when x is not an exact 
integer, or an ‘out-of-range’ error when it doesn’t fit the given range.

The functions scm_to_long_long, scm_to_ulonglong, scm_to_int64, and scm_to_ 
uint64 are only available when the corresponding types are.

SCM scm_from_char (char x)  
[C Function]

SCM scm_from_schar (signed char x)  
[C Function]
SCM scm_from_uchar (unsigned char x)  
SCM scm_from_short (short x)  
SCM scm_from_ushort (unsigned short x)  
SCM scm_from_int (int x)  
SCM scm_from_uint (unsigned int x)  
SCM scm_from_long (long x)  
SCM scm_from_ulong (unsigned long x)  
SCM scm_from_long_long (long long x)  
SCM scm_from_ulong_long (unsigned long long x)  
SCM scm_from_size_t (size_t x)  
SCM scm_from_ssize_t (ssize_t x)  
SCM scm_from_uintptr_t (uintptr_t x)  
SCM scm_from_sptrdiff_t (scm_t_ptrdiff x)  
SCM scm_from_int8 (scm_t_int8 x)  
SCM scm_from_uint8 (scm_t_uint8 x)  
SCM scm_from_int16 (scm_t_int16 x)  
SCM scm_from_uint16 (scm_t_uint16 x)  
SCM scm_from_int32 (scm_t_int32 x)  
SCM scm_from_uint32 (scm_t_uint32 x)  
SCM scm_from_int64 (scm_t_int64 x)  
SCM scm_from_uint64 (scm_t_uint64 x)  
SCM scm_from_intmax (scm_t_intmax x)  
SCM scm_from_uintmax (scm_t_uintmax x)  
SCM scm_from_intptr_t (scm_t_intptr x)  
SCM scm_from_uintptr_t (scm_t_uintptr x)  

Return the SCM value that represents the integer x. These functions will always succeed and will always return an exact number.

void scm_to_mpz (SCM val, mpz_t rop)  
Assign val to the multiple precision integer rop. val must be an exact integer, otherwise an error will be signalled. rop must have been initialized with mpz_init before this function is called. When rop is no longer needed the occupied space must be freed with mpz_clear. See Section “Initializing Integers” in GNU MP Manual, for details.

SCM scm_from_mpz (mpz_t val)  
Return the SCM value that represents val.

6.6.2.3 Real and Rational Numbers
Mathematically, the real numbers are the set of numbers that describe all possible points along a continuous, infinite, one-dimensional line. The rational numbers are the set of all numbers that can be written as fractions \( \frac{p}{q} \), where \( p \) and \( q \) are integers. All rational numbers are also real, but there are real numbers that are not rational, for example \( \sqrt{2} \) and \( \pi \).

Guile can represent both exact and inexact rational numbers, but it cannot represent precise finite irrational numbers. Exact rationals are represented by storing the numer-
tor and denominator as two exact integers. Inexact rationals are stored as floating point numbers using the C type double.

Exact rationals are written as a fraction of integers. There must be no whitespace around the slash:

\[
\begin{align*}
1/2 \\
-22/7
\end{align*}
\]

Even though the actual encoding of inexact rationals is in binary, it may be helpful to think of it as a decimal number with a limited number of significant figures and a decimal point somewhere, since this corresponds to the standard notation for non-whole numbers. For example:

\[
\begin{align*}
0.34 \\
-0.00000142857931198 \\
-5648394822220000000.0 \\
4.0
\end{align*}
\]

The limited precision of Guile’s encoding means that any finite “real” number in Guile can be written in a rational form, by multiplying and then dividing by sufficient powers of 10 (or in fact, 2). For example, ‘-0.00000142857931198’ is the same as \(-142857931198\) divided by \(1000000000000000.0\). In Guile’s current incarnation, therefore, the rational? and real? predicates are equivalent for finite numbers.

Dividing by an exact zero leads to a error message, as one might expect. However, dividing by an inexact zero does not produce an error. Instead, the result of the division is either plus or minus infinity, depending on the sign of the divided number and the sign of the zero divisor (some platforms support signed zeroes ‘-0.0’ and ‘+0.0’; ‘0.0’ is the same as ‘+0.0’).

Dividing zero by an inexact zero yields a NaN (‘not a number’) value, although they are actually considered numbers by Scheme. Attempts to compare a NaN value with any number (including itself) using =, <, >, <= or >= always returns #f. Although a NaN value is not = to itself, it is both eqv? and equal? to itself and other NaN values. However, the preferred way to test for them is by using nan?.

The real NaN values and infinities are written ‘+nan.0’, ‘+inf.0’ and ‘-inf.0’. This syntax is also recognized by read as an extension to the usual Scheme syntax. These special values are considered by Scheme to be inexact real numbers but not rational. Note that non-real complex numbers may also contain infinities or NaN values in their real or imaginary parts. To test a real number to see if it is infinite, a NaN value, or neither, use inf?, nan?, or finite?, respectively. Every real number in Scheme belongs to precisely one of those three classes.

On platforms that follow IEEE 754 for their floating point arithmetic, the ‘+inf.0’, ‘-inf.0’, and ‘+nan.0’ values are implemented using the corresponding IEEE 754 values. They behave in arithmetic operations like IEEE 754 describes it, i.e., \((= +nan.0 +nan.0) \Rightarrow #f\).

real? obj

[Scheme Procedure]

scm_real_p (obj)

[C Function]

Return #t if obj is a real number, else #f. Note that the sets of integer and rational values form subsets of the set of real numbers, so the predicate will also be fulfilled if obj is an integer number or a rational number.
rational? x

scm_rational_p (x)

Return #t if x is a rational number, #f otherwise. Note that the set of integer values forms a subset of the set of rational numbers, i.e. the predicate will also be fulfilled if x is an integer number.

rationalize x eps

scm_rationalize (x, eps)

Returns the simplest rational number differing from x by no more than eps.

As required by R5RS, rationalize only returns an exact result when both its arguments are exact. Thus, you might need to use inexact->exact on the arguments.

   (rationalize (inexact->exact 1.2) 1/100)
   ⇒ 6/5

inf? x

scm_inf_p (x)

Return #t if the real number x is '+inf.0' or '-inf.0'. Otherwise return #f.

nan? x

scm_nan_p (x)

Return #t if the real number x is '+nan.0', or #f otherwise.

finite? x

scm_finite_p (x)

Return #t if the real number x is neither infinite nor a NaN, #f otherwise.

nan

scm_nan ()

Return '+nan.0', a NaN value.

inf

scm_inf ()

Return '+inf.0', positive infinity.

numerator x

scm_numerator (x)

Return the numerator of the rational number x.

denominator x

scm_denominator (x)

Return the denominator of the rational number x.

int scm_is_real (SCM val)

int scm_is_rational (SCM val)

Equivalent to scm_is_true (scm_real_p (val)) and scm_is_true (scm_rational_p (val)), respectively.

double scm_to_double (SCM val)

Returns the number closest to val that is representable as a double. Returns infinity for a val that is too large in magnitude. The argument val must be a real number.
SCM scm_from_double (double val)  

Return the SCM value that represents val. The returned value is inexact according to the predicate inexact?, but it will be exactly equal to val.

6.6.2.4 Complex Numbers

Complex numbers are the set of numbers that describe all possible points in a two-dimensional space. The two coordinates of a particular point in this space are known as the real and imaginary parts of the complex number that describes that point.

In Guile, complex numbers are written in rectangular form as the sum of their real and imaginary parts, using the symbol i to indicate the imaginary part.

\[ 3+4i \quad \Rightarrow \quad 3.0+4.0i \]

\[(\ast\ 3-8i\ 2.3+0.3i)\]

\[\Rightarrow\quad 9.3-17.5i \]

Polar form can also be used, with an ‘@’ between magnitude and angle,

\[1\@3.141592\quad \Rightarrow\quad -1.0\quad (approx)\]

\[-1\@1.57079\quad \Rightarrow\quad 0.0-1.0i\quad (approx)\]

Guile represents a complex number as a pair of inexact reals, so the real and imaginary parts of a complex number have the same properties of inexactness and limited precision as single inexact real numbers.

Note that each part of a complex number may contain any inexact real value, including the special values ‘+nan.0’, ‘+inf.0’ and ‘-inf.0’, as well as either of the signed zeroes ‘0.0’ or ‘-0.0’.

complex? z  

scm_complex_p (z)  

Return #t if z is a complex number, #f otherwise. Note that the sets of real, rational and integer values form subsets of the set of complex numbers, i.e. the predicate will also be fulfilled if z is a real, rational or integer number.

int scm_is_complex (SCM val)  

Equivalent to scm_is_true (scm_complex_p (val)).

6.6.2.5 Exact and Inexact Numbers

R5RS requires that, with few exceptions, a calculation involving inexact numbers always produces an inexact result. To meet this requirement, Guile distinguishes between an exact integer value such as ‘5’ and the corresponding inexact integer value which, to the limited precision available, has no fractional part, and is printed as ‘5.0’. Guile will only convert the latter value to the former when forced to do so by an invocation of the inexact->exact procedure.

The only exception to the above requirement is when the values of the inexact numbers do not affect the result. For example (expt n 0) is ‘1’ for any value of n, therefore (expt 5.0 0) is permitted to return an exact ‘1’.
**exact? z**  

**scm_exact_p (z)**  

Return #t if the number z is exact, #f otherwise.

- **(exact? 2)**  
  ⇒ #t

- **(exact? 0.5)**  
  ⇒ #f

- **(exact? (/ 2))**  
  ⇒ #t

**int scm_is_exact (SCM z)**  

Return a 1 if the number z is exact, and 0 otherwise. This is equivalent to scm_is_true (scm_exact_p (z)).

An alternate approach to testing the exactness of a number is to use scm_is_signed_integer or scm_is_unsigned_integer.

**inexact? z**  

**scm_inexact_p (z)**  

Return #t if the number z is inexact, #f otherwise.

**int scm_is_inexact (SCM z)**  

Return a 1 if the number z is inexact, and 0 otherwise. This is equivalent to scm_is_true (scm_inexact_p (z)).

**inexact->exact z**  

**scm_inexact_to_exact (z)**  

Return an exact number that is numerically closest to z, when there is one. For inexact rationals, Guile returns the exact rational that is numerically equal to the inexact rational. Inexact complex numbers with a non-zero imaginary part can not be made exact.

- **(inexact->exact 0.5)**  
  ⇒ 1/2

The following happens because 12/10 is not exactly representable as a double (on most platforms). However, when reading a decimal number that has been marked exact with the “#e” prefix, Guile is able to represent it correctly.

- **(inexact->exact 1.2)**  
  ⇒ 5404319552844595/4503599627370496

  #e1.2  
  ⇒ 6/5

**exact->inexact z**  

**scm_exact_to_inexact (z)**  

Convert the number z to its inexact representation.
6.6.2.6 Read Syntax for Numerical Data

The read syntax for integers is a string of digits, optionally preceded by a minus or plus character, a code indicating the base in which the integer is encoded, and a code indicating whether the number is exact or inexact. The supported base codes are:

- `#b` - the integer is written in binary (base 2)
- `#B` - the integer is written in binary (base 2)
- `#o` - the integer is written in octal (base 8)
- `#O` - the integer is written in octal (base 8)
- `#d` - the integer is written in decimal (base 10)
- `#D` - the integer is written in decimal (base 10)
- `#x` - the integer is written in hexadecimal (base 16)
- `#X` - the integer is written in hexadecimal (base 16)

If the base code is omitted, the integer is assumed to be decimal. The following examples show how these base codes are used.

- `-13` ⇒ `-13`
- `#d-13` ⇒ `-13`
- `#x-13` ⇒ `-19`
- `#b+1101` ⇒ `13`
- `#o377` ⇒ `255`

The codes for indicating exactness (which can, incidentally, be applied to all numerical values) are:

- `#e` - the number is exact
- `#E` - the number is exact
- `#i` - the number is inexact
- `#I` - the number is inexact.

If the exactness indicator is omitted, the number is exact unless it contains a radix point. Since Guile can not represent exact complex numbers, an error is signalled when asking for them.

- `(exact? 1.2)` ⇒ `#f`
- `(exact? #e1.2)` ⇒ `#t`
(exact? #e+i)  
ERROR: Wrong type argument

Guile also understands the syntax ‘+inf.0’ and ‘-inf.0’ for plus and minus infinity, respectively. The value must be written exactly as shown, that is, they always must have a sign and exactly one zero digit after the decimal point. It also understands ‘+nan.0’ and ‘-nan.0’ for the special ‘not-a-number’ value. The sign is ignored for ‘not-a-number’ and the value is always printed as ‘+nan.0’.

### 6.6.2.7 Operations on Integer Values

**odd? n**  
**scm_odd_p (n)**  
Return #t if n is an odd number, #f otherwise.

**even? n**  
**scm_even_p (n)**  
Return #t if n is an even number, #f otherwise.

**quotient n d**  
**remainder n d**  
**scm_quotient (n, d)**  
**scm_remainder (n, d)**  
Return the quotient or remainder from n divided by d. The quotient is rounded towards zero, and the remainder will have the same sign as n. In all cases quotient and remainder satisfy n = q * d + r.

- (remainder 13 4) ⇒ 1
- (remainder -13 4) ⇒ -1

See also **truncate-quotient**, **truncate-remainder** and related operations in Section 6.6.2.11 [Arithmetic], page 119.

**modulo n d**  
**scm_modulo (n, d)**  
Return the remainder from n divided by d, with the same sign as d.

- (modulo 13 4) ⇒ 1
- (modulo -13 4) ⇒ 3
- (modulo 13 -4) ⇒ -3
- (modulo -13 -4) ⇒ -1

See also **floor-quotient**, **floor-remainder** and related operations in Section 6.6.2.11 [Arithmetic], page 119.

**gcd x...**  
**scm_gcd (x, y)**  
Return the greatest common divisor of all arguments. If called without arguments, 0 is returned.

The C function **scm_gcd** always takes two arguments, while the Scheme function can take an arbitrary number.
lcm x...  
scm_lcm (x, y)  
Return the least common multiple of the arguments. If called without arguments, 1 is returned.

The C function scm_lcm always takes two arguments, while the Scheme function can take an arbitrary number.

modulo-expt n k m  
scm_modulo_expt (n, k, m)  
Return n raised to the integer exponent k, modulo m.

(modulo-expt 2 3 5)  
⇒ 3

effect-integer-sqrt k  
void scm_effect_integer_sqrt (SCM k, SCM *s, SCM *r)  
Return two exact non-negative integers s and r such that k = s^2 + r and s^2 <= k < (s + 1)^2. An error is raised if k is not an exact non-negative integer.

(exact-integer-sqrt 10)  
⇒ 3 and 1

6.6.2.8 Comparison Predicates

The C comparison functions below always takes two arguments, while the Scheme functions can take an arbitrary number. Also keep in mind that the C functions return one of the Scheme boolean values SCM_BOOL_T or SCM_BOOL_F which are both true as far as C is concerned. Thus, always write scm_is_true (scm_num_eq_p (x, y)) when testing the two Scheme numbers x and y for equality, for example.

=  
scm_num_eq_p (x, y)  
Return #t if all parameters are numerically equal.

<>  
scm_less_p (x, y)  
Return #t if the list of parameters is monotonically increasing.

>  
scm_gr_p (x, y)  
Return #t if the list of parameters is monotonically decreasing.

<=  
scm_leq_p (x, y)  
Return #t if the list of parameters is monotonically non-decreasing.

>=  
scm_geq_p (x, y)  
Return #t if the list of parameters is monotonically non-increasing.

zero? z  
scm_zero_p (z)  
Return #t if z is an exact or inexact number equal to zero.
positive? x

scm_positive_p (x)

Return #t if x is an exact or inexact number greater than zero.

negative? x

scm_negative_p (x)

Return #t if x is an exact or inexact number less than zero.

6.6.2.9 Converting Numbers To and From Strings

The following procedures read and write numbers according to their external representation as defined by R5RS (see Section “Lexical structure” in The Revised’5 Report on the Algorithmic Language Scheme). See Section 6.25.4 [Number Input and Output], page 462, for locale-dependent number parsing.

number->string n [radix]

scm_number_to_string (n, radix)

Return a string holding the external representation of the number n in the given radix. If n is inexact, a radix of 10 will be used.

string->number string [radix]

scm_string_to_number (string, radix)

Return a number of the maximally precise representation expressed by the given string. radix must be an exact integer, either 2, 8, 10, or 16. If supplied, radix is a default radix that may be overridden by an explicit radix prefix in string (e.g. "#o177"). If radix is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number, then string->number returns #f.

SCM scm_c_locale_stringn_to_number (const char *string, size_t len, unsigned radix)

As per string->number above, but taking a C string, as pointer and length. The string characters should be in the current locale encoding (locale in the name refers only to that, there’s no locale-dependent parsing).

6.6.2.10 Complex Number Operations

make-rectangular real_part imaginary_part

scm_make_rectangular (real_part, imaginary_part)

Return a complex number constructed of the given real-part and imaginary-part parts.

make-polar mag ang

scm_make_polar (mag, ang)

Return the complex number mag * e^(i * ang).

real-part z

scm_real_part (z)

Return the real part of the number z.

imag-part z

scm_imag_part (z)

Return the imaginary part of the number z.
### 6.6.2.11 Arithmetic Functions

The C arithmetic functions below always takes two arguments, while the Scheme functions can take an arbitrary number. When you need to invoke them with just one argument, for example to compute the equivalent of \((- x)\), pass `SCM_UNDEFINED` as the second one: `scm_difference (x, SCM_UNDEFINED)`.

**+ z1 ...**  
`scm_sum (z1, z2)`  
Return the sum of all parameter values. Return 0 if called without any parameters.

**- z1 z2 ...**  
`scm_difference (z1, z2)`  
If called with one argument `z1`, \(-z1\) is returned. Otherwise the sum of all but the first argument are subtracted from the first argument.

**\(\ast\) z1 ...**  
`scm_product (z1, z2)`  
Return the product of all arguments. If called without arguments, 1 is returned.

**/ z1 z2 ...**  
`scm_divide (z1, z2)`  
Divide the first argument by the product of the remaining arguments. If called with one argument `z1`, \(1/z1\) is returned.

**1+ z**  
`scm_oneplus (z)`  
Return `z + 1`.  

---

**magnitude z**  
`scm_magnitude (z)`  
Return the magnitude of the number `z`. This is the same as `abs` for real arguments, but also allows complex numbers.

**angle z**  
`scm_angle (z)`  
Return the angle of the complex number `z`.

**SCM scm_c_make_rectangular (double re, double im)**  
**SCM scm_c_make_polar (double x, double y)**  
Like `scm_make_rectangular` or `scm_make_polar`, respectively, but these functions take doubles as their arguments.

**double scm_c_real_part (z)**  
**double scm_c_imag_part (z)**  
Returns the real or imaginary part of `z` as a `double`.

**double scm_c_magnitude (z)**  
**double scm_c_angle (z)**  
Returns the magnitude or angle of `z` as a `double`. 
1- \(z\)

**scm_oneminus** (\(z\))

Return \(z - 1\).

**abs** \(x\)

**scm_abs** (\(x\))

Return the absolute value of \(x\).

\(x\) must be a number with zero imaginary part. To calculate the magnitude of a complex number, use **magnitude** instead.

**max** \(x_1 x_2 \ldots\)

**scm_max** (\(x_1, x_2\))

Return the maximum of all parameter values.

**min** \(x_1 x_2 \ldots\)

**scm_min** (\(x_1, x_2\))

Return the minimum of all parameter values.

**truncate** \(x\)

**scm_truncate_number** (\(x\))

Round the inexact number \(x\) towards zero.

**round** \(x\)

**scm_round_number** (\(x\))

Round the inexact number \(x\) to the nearest integer. When exactly halfway between two integers, round to the even one.

**floor** \(x\)

**scm_floor** (\(x\))

Round the number \(x\) towards minus infinity.

**ceiling** \(x\)

**scm_ceiling** (\(x\))

Round the number \(x\) towards infinity.

**double** \(x\)

**scm_c_truncate** (\(x\))

**scm_c_round** (\(x\))

Like **scm_truncate_number** or **scm_round_number**, respectively, but these functions take and return **double** values.

**euclidean/** **x y**

**euclidean-quotient** **x y**

**euclidean-remainder** **x y**

**void** **scm_euclidean_divide** (**SCM** \(x\), **SCM** \(y\), **SCM** \(*q\), **SCM** \(*r\))

**SCM** **scm_euclidean_quotient** (**SCM** \(x\), **SCM** \(y\))

**SCM** **scm_euclidean_remainder** (**SCM** \(x\), **SCM** \(y\))

These procedures accept two real numbers \(x\) and \(y\), where the divisor \(y\) must be non-zero. **euclidean-quotient** returns the integer \(q\) and **euclidean-remainder** returns the real number \(r\) such that \(x = q \cdot y + r\) and \(0 \leq r < |y|\). **euclidean/** returns
both \( q \) and \( r \), and is more efficient than computing each separately. Note that when \( y > 0 \), \texttt{euclidean-quotient} returns \( \text{floor}(x/y) \), otherwise it returns \( \text{ceiling}(x/y) \). Note that these operators are equivalent to the R6RS operators \texttt{div}, \texttt{mod}, and \texttt{div-and-mod}.

\[
\begin{align*}
\text{(euclidean-quotient 123 10)} & \Rightarrow 12 \\
\text{(euclidean-remainder 123 10)} & \Rightarrow 3 \\
\text{(euclidean/ 123 10)} & \Rightarrow 12 \text{ and } 3 \\
\text{(euclidean/ 123 -10)} & \Rightarrow -12 \text{ and } 3 \\
\text{(euclidean/ -123 10)} & \Rightarrow -13 \text{ and } 7 \\
\text{(euclidean/ -123 -10)} & \Rightarrow 13 \text{ and } 7 \\
\text{(euclidean/ -123.2 -63.5)} & \Rightarrow 2.0 \text{ and } 3.8 \\
\text{(euclidean/ -16/3 -10/7)} & \Rightarrow -3 \text{ and } 22/21 \\
\end{align*}
\]

\textbf{floor/ \( x \ y \) [Scheme Procedure]}
\textbf{floor-quotient \( x \ y \) [Scheme Procedure]}
\textbf{floor-remainder \( x \ y \) [Scheme Procedure]}

\begin{verbatim}
void scm_floor_divide (SCM x, SCM y, SCM *q, SCM *r) [C Function]
SCM scm_floor_quotient (x, y) [C Function]
SCM scm_floor_remainder (x, y) [C Function]
\end{verbatim}

These procedures accept two real numbers \( x \) and \( y \), where the divisor \( y \) must be non-zero. \texttt{floor-quotient} returns the integer \( q \) and \texttt{floor-remainder} returns the real number \( r \) such that \( q = \text{floor}(x/y) \) and \( x = q \times y + r \). \texttt{floor/} returns both \( q \) and \( r \), and is more efficient than computing each separately. Note that \( r \), if non-zero, will have the same sign as \( y \).

When \( x \) and \( y \) are integers, \texttt{floor-remainder} is equivalent to the R5RS integer-only operator \texttt{modulo}.

\[
\begin{align*}
\text{(floor-quotient 123 10)} & \Rightarrow 12 \\
\text{(floor-remainder 123 10)} & \Rightarrow 3 \\
\text{(floor/ 123 10)} & \Rightarrow 12 \text{ and } 3 \\
\text{(floor/ 123 -10)} & \Rightarrow -13 \text{ and } -7 \\
\text{(floor/ -123 10)} & \Rightarrow -13 \text{ and } 7 \\
\text{(floor/ -123 -10)} & \Rightarrow 12 \text{ and } -3 \\
\text{(floor/ -123.2 -63.5)} & \Rightarrow 1.0 \text{ and } -59.7 \\
\text{(floor/ -16/3 -10/7)} & \Rightarrow -4 \text{ and } -8/21 \\
\end{align*}
\]

\textbf{ceiling/ \( x \ y \) [Scheme Procedure]}
\textbf{ceiling-quotient \( x \ y \) [Scheme Procedure]}
\textbf{ceiling-remainder \( x \ y \) [Scheme Procedure]}

\begin{verbatim}
void scm_ceiling_divide (SCM x, SCM y, SCM *q, SCM *r) [C Function]
SCM scm_ceiling_quotient (x, y) [C Function]
SCM scm_ceiling_remainder (x, y) [C Function]
\end{verbatim}

These procedures accept two real numbers \( x \) and \( y \), where the divisor \( y \) must be non-zero. \texttt{ceiling-quotient} returns the integer \( q \) and \texttt{ceiling-remainder} returns the real number \( r \) such that \( q = \text{ceiling}(x/y) \) and \( x = q \times y + r \). \texttt{ceiling/} returns both \( q \) and \( r \), and is more efficient than computing each separately. Note that \( r \), if non-zero, will have the opposite sign of \( y \).

\[
\text{(ceiling-quotient 123 10)} \Rightarrow 13
\]
(ceiling-remainder 123 10) ⇒ -7
(ceiling/ 123 10) ⇒ 13 and -7
(ceiling/ 123 -10) ⇒ -12 and 3
(ceiling/ -123 10) ⇒ -12 and -3
(ceiling/ -123 -10) ⇒ 13 and 7
(ceiling/ -123.2 -63.5) ⇒ 2.0 and 3.8
(ceiling/ 16/3 -10/7) ⇒ -3 and 22/21

(truncate/ x y) [Scheme Procedure]
(truncate-quotient x y) [Scheme Procedure]
(truncate-remainder x y) [Scheme Procedure]
void scm_truncate_divide (SCM x, SCM y, SCM *q, SCM *r) [C Function]
SCM scm_truncate_quotient (x, y) [C Function]
SCM scm_truncate_remainder (x, y) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. truncate-quotient returns the integer q and truncate-remainder returns the real number r such that q is x/y rounded toward zero, and x = q * y + r. truncate/ returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the same sign as x.

When x and y are integers, these operators are equivalent to the R5RS integer-only operators quotient and remainder.

(truncate-quotient 123 10) ⇒ 12
(truncate-remainder 123 10) ⇒ 3
(truncate/ 123 10) ⇒ 12 and 3
(truncate/ 123 -10) ⇒ -12 and 3
(truncate/ -123 10) ⇒ -12 and -3
(truncate/ -123 -10) ⇒ 12 and -3
(truncate/ -123.2 -63.5) ⇒ 1.0 and -59.7
(truncate/ 16/3 -10/7) ⇒ -3 and 22/21

centered/ x y [Scheme Procedure]
centered-quotient x y [Scheme Procedure]
centered-remainder x y [Scheme Procedure]
void scm_centered_divide (SCM x, SCM y, SCM *q, SCM *r) [C Function]
SCM scm_centered_quotient (SCM x, SCM y) [C Function]
SCM scm_centered_remainder (SCM x, SCM y) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. centered-quotient returns the integer q and centered-remainder returns the real number r such that x = q * y + r and -|y/2| <= r < |y/2|. centered/ returns both q and r, and is more efficient than computing each separately.

Note that centered-quotient returns x/y rounded to the nearest integer. When x/y lies exactly halfway between two integers, the tie is broken according to the sign of y. If y > 0, ties are rounded toward positive infinity, otherwise they are rounded toward negative infinity. This is a consequence of the requirement that -|y/2| <= r < |y/2|. Note that these operators are equivalent to the R6RS operators div0, mod0, and div0-and-mod0.

(centered-quotient 123 10) ⇒ 12
(centered-remainder 123 10) ⇒ 3
(centered/ 123 10) ⇒ 12 and 3
(centered/ 123 -10) ⇒ -12 and 3
(centered/ -123 10) ⇒ -12 and -3
(centered/ -123 -10) ⇒ 12 and -3
(centered/ 125 10) ⇒ 13 and -5
(centered/ 127 10) ⇒ 13 and -3
(centered/ 135 10) ⇒ 14 and -5
(centered/ -123.2 -63.5) ⇒ 2.0 and 3.8
(centered/ 16/3 -10/7) ⇒ -4 and -8/21

round/ x y [Scheme Procedure]
round-quotient x y [Scheme Procedure]
round-remainder x y [Scheme Procedure]
void scm_round_divide (SCM x, SCM y, SCM *q, SCM *r) [C Function]
SCM scm_round_quotient (x, y) [C Function]
SCM scm_round_remainder (x, y) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. round-quotient returns the integer q and round-remainder returns the real number r such that x = q * y + r and q is x/y rounded to the nearest integer, with ties going to the nearest even integer. round/ returns both q and r, and is more efficient than computing each separately.

Note that round/ and centered/ are almost equivalent, but their behavior differs when x/y lies exactly half-way between two integers. In this case, round/ chooses the nearest even integer, whereas centered/ chooses in such a way to satisfy the constraint -|y/2| <= r < |y/2|, which is stronger than the corresponding constraint for round/. -|y/2| <= r <= |y/2|. In particular, when x and y are integers, the number of possible remainders returned by centered/ is |y|, whereas the number of possible remainders returned by round/ is |y| + 1 when y is even.

(round-quotient 123 10) ⇒ 12
(round-remainder 123 10) ⇒ 3
(round/ 123 10) ⇒ 12 and 3
(round/ 123 -10) ⇒ -12 and 3
(round/ -123 10) ⇒ -12 and -3
(round/ -123 -10) ⇒ 12 and -3
(round/ 125 10) ⇒ 12 and 5
(round/ 127 10) ⇒ 13 and -3
(round/ 135 10) ⇒ 14 and -5
(round/ -123.2 -63.5) ⇒ 2.0 and 3.8
(round/ 16/3 -10/7) ⇒ -4 and -8/21

6.6.2.12 Scientific Functions
The following procedures accept any kind of number as arguments, including complex numbers.
\texttt{sqrt} \ z \quad \text{[Scheme Procedure]}

Return the square root of \( z \). Of the two possible roots (positive and negative), the one with a positive real part is returned, or if that’s zero then a positive imaginary part. Thus,

- \( (\text{sqrt} \ 9.0) \Rightarrow 3.0 \)
- \( (\text{sqrt} \ -9.0) \Rightarrow 0.0+3.0i \)
- \( (\text{sqrt} \ 1.0+1.0i) \Rightarrow 1.09868411346781+0.455089860562227i \)
- \( (\text{sqrt} \ -1.0-1.0i) \Rightarrow 0.455089860562227-1.09868411346781i \)

\texttt{expt} \ z1 \ z2 \quad \text{[Scheme Procedure]}

Return \( z1 \) raised to the power of \( z2 \).

\texttt{sin} \ z \quad \text{[Scheme Procedure]}

Return the sine of \( z \).

\texttt{cos} \ z \quad \text{[Scheme Procedure]}

Return the cosine of \( z \).

\texttt{tan} \ z \quad \text{[Scheme Procedure]}

Return the tangent of \( z \).

\texttt{asin} \ z \quad \text{[Scheme Procedure]}

Return the arcsine of \( z \).

\texttt{acos} \ z \quad \text{[Scheme Procedure]}

Return the arccosine of \( z \).

\texttt{atan} \ z \quad \text{[Scheme Procedure]}

\texttt{atan} \ y \ x \quad \text{[Scheme Procedure]}

Return the arctangent of \( z \), or of \( y/x \).

\texttt{exp} \ z \quad \text{[Scheme Procedure]}

Return \( e \) to the power of \( z \), where \( e \) is the base of natural logarithms (2.71828...).

\texttt{log} \ z \quad \text{[Scheme Procedure]}

Return the natural logarithm of \( z \).

\texttt{log10} \ z \quad \text{[Scheme Procedure]}

Return the base 10 logarithm of \( z \).

\texttt{sinh} \ z \quad \text{[Scheme Procedure]}

Return the hyperbolic sine of \( z \).

\texttt{cosh} \ z \quad \text{[Scheme Procedure]}

Return the hyperbolic cosine of \( z \).

\texttt{tanh} \ z \quad \text{[Scheme Procedure]}

Return the hyperbolic tangent of \( z \).

\texttt{asinh} \ z \quad \text{[Scheme Procedure]}

Return the hyperbolic arcsine of \( z \).
acosh \( z \)  

Return the hyperbolic arccosine of \( z \).

atanh \( z \)  

Return the hyperbolic arctangent of \( z \).

### 6.6.2.13 Bitwise Operations

For the following bitwise functions, negative numbers are treated as infinite precision two's-complements. For instance \(-6\) is bits \(...111010\), with infinitely many ones on the left. It can be seen that adding 6 (binary 110) to such a bit pattern gives all zeros.

**logand** \( n_1 \ n_2 \ldots \)  

scm_logand \((n_1, n_2)\)  

Return the bitwise AND of the integer arguments.

\[
\begin{align*}
(\text{logand}) & \Rightarrow -1 \\
(\text{logand} 7) & \Rightarrow 7 \\
(\text{logand} \ #b111 \ #b011 \ #b001) & \Rightarrow 1
\end{align*}
\]

**logior** \( n_1 \ n_2 \ldots \)  

scm_logior \((n_1, n_2)\)  

Return the bitwise OR of the integer arguments.

\[
\begin{align*}
(\text{logior}) & \Rightarrow 0 \\
(\text{logior} 7) & \Rightarrow 7 \\
(\text{logior} \ #b000 \ #b001 \ #b011) & \Rightarrow 3
\end{align*}
\]

**logxor** \( n_1 \ n_2 \ldots \)  

scm_logxor \((n_1, n_2)\)  

Return the bitwise XOR of the integer arguments. A bit is set in the result if it is set in an odd number of arguments.

\[
\begin{align*}
(\text{logxor}) & \Rightarrow 0 \\
(\text{logxor} 7) & \Rightarrow 7 \\
(\text{logxor} \ #b000 \ #b001 \ #b011) & \Rightarrow 2 \\
(\text{logxor} \ #b000 \ #b001 \ #b011 \ #b011) & \Rightarrow 1
\end{align*}
\]

**lognot** \( n \)  

scm_lognot \((n)\)  

Return the integer which is the ones-complement of the integer argument, ie. each 0 bit is changed to 1 and each 1 bit to 0.

\[
\begin{align*}
\text{(number->string \ (lognot \ #b10000000) \ 2)} & \Rightarrow "-10000001" \\
\text{(number->string \ (lognot \ #b0) \ 2)} & \Rightarrow "-1"
\end{align*}
\]

**logtest** \( j \ k \)  

scm_logtest \((j, k)\)  

Test whether \( j \) and \( k \) have any 1 bits in common. This is equivalent to \((\text{not (zero? (logand j k))})\), but without actually calculating the \text{logand}, just testing for non-zero.

\[
(\text{logtest} \ #b0100 \ #b1011) \Rightarrow \#f
\]
(logtest #b0100 #b0111) ⇒ #t

logbit?  index  j  

scm_logbit_p (index, j)  

Test whether bit number index in j is set. index starts from 0 for the least significant bit.

(logbit? 0 #b1101) ⇒ #t  
(logbit? 1 #b1101) ⇒ #f  
(logbit? 2 #b1101) ⇒ #t  
(logbit? 3 #b1101) ⇒ #t  
(logbit? 4 #b1101) ⇒ #f

ash  n  count  

scm_ash (n, count)  

Return floor(n * \(2^{\text{count}}\)). n and count must be exact integers.  

With n viewed as an infinite-precision twos-complement integer, ash means a left shift introducing zero bits when count is positive, or a right shift dropping bits when count is negative. This is an “arithmetic” shift.

(number->string (ash #b1 3) 2) ⇒ "1000"  
(number->string (ash #b1010 -1) 2) ⇒ "101"

;; -23 is bits ...11101001, -6 is bits ...111010  
(ash -23 -2) ⇒ -6

round-ash  n  count  

scm_round_ash (n, count)  

Return round(n * \(2^{\text{count}}\)). n and count must be exact integers.  

With n viewed as an infinite-precision twos-complement integer, round-ash means a left shift introducing zero bits when count is positive, or a right shift rounding to the nearest integer (with ties going to the nearest even integer) when count is negative. This is a rounded “arithmetic” shift.

(number->string (round-ash #b1 3) 2) ⇒ "1000"  
(number->string (round-ash #b1010 -1) 2) ⇒ "101"  
(number->string (round-ash #b1010 -2) 2) ⇒ "10"  
(number->string (round-ash #b1011 -2) 2) ⇒ "11"  
(number->string (round-ash #b1101 -2) 2) ⇒ "100"

logcount  n  

scm_logcount (n)  

Return the number of bits in integer n. If n is positive, the 1-bits in its binary representation are counted. If negative, the 0-bits in its two’s-complement binary representation are counted. If zero, 0 is returned.

(logcount #b10101010) ⇒ 4  
(logcount 0) ⇒ 0
\[(\logcount -2) \Rightarrow 1\]

**integer-length n**  
_SCHEME Procedure_  
**scm_integer_length (n)**  
_C Function_  
Return the number of bits necessary to represent \(n\).  
For positive \(n\) this is how many bits to the most significant one bit. For negative \(n\) it’s how many bits to the most significant zero bit in two's complement form.

\[
\begin{align*}
(\text{integer-length } #b10101010) & \Rightarrow 8 \\
(\text{integer-length } #b1111) & \Rightarrow 4 \\
(\text{integer-length } 0) & \Rightarrow 0 \\
(\text{integer-length } -1) & \Rightarrow 0 \\
(\text{integer-length } -256) & \Rightarrow 8 \\
(\text{integer-length } -257) & \Rightarrow 9
\end{align*}
\]

**integer-expt n k**  
_SCHEME Procedure_  
**scm_integer_expt (n, k)**  
_C Function_  
Return \(n\) raised to the power \(k\). \(k\) must be an exact integer, \(n\) can be any number.  
Negative \(k\) is supported, and results in \(1/n^{|k|}\) in the usual way. \(n^0\) is 1, as usual, and that includes \(0^0\) is 1.

\[
\begin{align*}
(\text{integer-expt } 2 5) & \Rightarrow 32 \\
(\text{integer-expt } -3 3) & \Rightarrow -27 \\
(\text{integer-expt } 5 -3) & \Rightarrow 1/125 \\
(\text{integer-expt } 0 0) & \Rightarrow 1
\end{align*}
\]

**bit-extract n start end**  
_SCHEME Procedure_  
**scm_bit_extract (n, start, end)**  
_C Function_  
Return the integer composed of the \(start\) (inclusive) through \(end\) (exclusive) bits of \(n\). The \(start\)th bit becomes the 0-th bit in the result.

\[
\begin{align*}
(\text{number->string } (\text{bit-extract } #b1101101010 0 4) 2) & \Rightarrow "1010" \\
(\text{number->string } (\text{bit-extract } #b1101101010 4 9) 2) & \Rightarrow "10110"
\end{align*}
\]

## 6.6.2.14 Random Number Generation

Pseudo-random numbers are generated from a random state object, which can be created with \texttt{seed->random-state} or \texttt{datum->random-state}. An external representation (i.e. one which can written with \texttt{write} and read with \texttt{read}) of a random state object can be obtained via \texttt{random-state->datum}. The \texttt{state} parameter to the various functions below is optional, it defaults to the state object in the \*random-state* variable.

**copy-random-state \[state\]**  
_SCHEME Procedure_  
**scm_copy_random_state (state)**  
_C Function_  
Return a copy of the random state \(state\).

**random \[n\] \[state\]**  
_SCHEME Procedure_  
**scm_random (n, state)**  
_C Function_  
Return a number in \([0, n)\).
Accepts a positive integer or real \( n \) and returns a number of the same type between zero (inclusive) and \( n \) (exclusive). The values returned have a uniform distribution.

\[
\text{random:exp} \ [\text{state}] \\
\text{scm_random_exp (state)} \\
\text{Return an inexact real in an exponential distribution with mean 1. For an exponential distribution with mean } u \text{ use } (* \ u \ (\text{random:exp})).
\]

\[
\text{random:hollow-sphere! \ vect} \ [\text{state}] \\
\text{scm_random_hollow_sphere_x (vect, state)} \\
\text{Fills vect with inexact real random numbers the sum of whose squares is equal to 1.0. Thinking of vect as coordinates in space of dimension } n = (\text{vector-length vect}), the coordinates are uniformly distributed over the surface of the unit n-sphere.
\]

\[
\text{random:normal} \ [\text{state}] \\
\text{scm_random_normal (state)} \\
\text{Return an inexact real in a normal distribution. The distribution used has mean 0 and standard deviation 1. For a normal distribution with mean } m \text{ and standard deviation } d \text{ use } (* \ m \ (* \ d \ (\text{random:normal}))).
\]

\[
\text{random:uniform} \ [\text{state}] \\
\text{scm_random_uniform (state)} \\
\text{Return a uniformly distributed inexact real random number in } [0,1).
\]

\[
\text{seed->random-state} \ seed \\
\text{scm_seed_to_random_state (seed)} \\
\text{Return a new random state using } seed.
\]

\[
\text{datum->random-state} \ datum \\
\text{scm_datum_to_random_state (datum)} \\
\text{Return a new random state from datum, which should have been obtained by random-state->datum.}
\]

\[
\text{random-state->datum} \ state \\
\text{scm_random_state_to_datum (state)} \\
\text{Return a datum representation of } state \text{ that may be written out and read back with the Scheme reader.}
\]
random-state-from-platform [Scheme Procedure]
s scm_random_state_from_platform () [C Function]
Construct a new random state seeded from a platform-specific source of entropy, appropriate for use in non-security-critical applications. Currently /dev/urandom is tried first, or else the seed is based on the time, date, process ID, an address from a freshly allocated heap cell, an address from the local stack frame, and a high-resolution timer if available.

*random-state* [Variable]
The global random state used by the above functions when the state parameter is not given.

Note that the initial value of *random-state* is the same every time Guile starts up. Therefore, if you don’t pass a state parameter to the above procedures, and you don’t set *random-state* to (seed->random-state your-seed), where your-seed is something that isn’t the same every time, you’ll get the same sequence of “random” numbers on every run.

For example, unless the relevant source code has changed, (map random (cdr (iota 30))), if the first use of random numbers since Guile started up, will always give:

(m map random (cdr (iota 19)))
⇒
(0 1 2 2 1 2 6 7 10 0 5 3 12 5 5 12)

To seed the random state in a sensible way for non-security-critical applications, do this during initialization of your program:

(set! *random-state* (random-state-from-platform))

6.6.3 Characters
In Scheme, there is a data type to describe a single character.

Defining what exactly a character is can be more complicated than it seems. Guile follows the advice of R6RS and uses The Unicode Standard to help define what a character is. So, for Guile, a character is anything in the Unicode Character Database.

The Unicode Character Database is basically a table of characters indexed using integers called 'code points’. Valid code points are in the ranges 0 to #xD7FF inclusive or #xE000 to #x10FFFF inclusive, which is about 1.1 million code points.

Any code point that has been assigned to a character or that has otherwise been given a meaning by Unicode is called a 'designated code point’. Most of the designated code points, about 200,000 of them, indicate characters, accents or other combining marks that modify other characters, symbols, whitespace, and control characters. Some are not characters but indicators that suggest how to format or display neighboring characters.

If a code point is not a designated code point – if it has not been assigned to a character by The Unicode Standard – it is a 'reserved code point’, meaning that they are reserved for future use. Most of the code points, about 800,000, are 'reserved code points’.

By convention, a Unicode code point is written as “U+XXXX” where “XXXX” is a hexadecimal number. Please note that this convenient notation is not valid code. Guile does not interpret “U+XXXX” as a character.
In Scheme, a character literal is written as \texttt{#\textit{name}} where \textit{name} is the name of the character that you want. Printable characters have their usual single character name; for example, \texttt{#\textsc{a}} is a lower case \textsc{a}.

Some of the code points are 'combining characters' that are not meant to be printed by themselves but are instead meant to modify the appearance of the previous character. For combining characters, an alternate form of the character literal is \texttt{#\} followed by U+25CC (a small, dotted circle), followed by the combining character. This allows the combining character to be drawn on the circle, not on the backslash of \texttt{#\}.

Many of the non-printing characters, such as whitespace characters and control characters, also have names.

The most commonly used non-printing characters have long character names, described in the table below.

<table>
<thead>
<tr>
<th>Character Name</th>
<th>Codepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{#\nul}</td>
<td>U+0000</td>
</tr>
<tr>
<td>\texttt{#\alarm}</td>
<td>U+0007</td>
</tr>
<tr>
<td>\texttt{#\backspace}</td>
<td>U+0008</td>
</tr>
<tr>
<td>\texttt{#\tab}</td>
<td>U+0009</td>
</tr>
<tr>
<td>\texttt{#\linefeed}</td>
<td>U+000A</td>
</tr>
<tr>
<td>\texttt{#\newline}</td>
<td>U+000A</td>
</tr>
<tr>
<td>\texttt{#\vtab}</td>
<td>U+000B</td>
</tr>
<tr>
<td>\texttt{#\page}</td>
<td>U+000C</td>
</tr>
<tr>
<td>\texttt{#\return}</td>
<td>U+000D</td>
</tr>
<tr>
<td>\texttt{#\esc}</td>
<td>U+001B</td>
</tr>
<tr>
<td>\texttt{#\space}</td>
<td>U+0020</td>
</tr>
<tr>
<td>\texttt{#\delete}</td>
<td>U+007F</td>
</tr>
</tbody>
</table>

There are also short names for all of the “C0 control characters” (those with code points below 32). The following table lists the short name for each character.

<table>
<thead>
<tr>
<th>Character Name</th>
<th>Codepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{#\nul}</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{#\soh}</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{#\stx}</td>
<td>2</td>
</tr>
<tr>
<td>\texttt{#\etx}</td>
<td>3</td>
</tr>
<tr>
<td>\texttt{#\eot}</td>
<td>4</td>
</tr>
<tr>
<td>\texttt{#\enq}</td>
<td>5</td>
</tr>
<tr>
<td>\texttt{#\ack}</td>
<td>6</td>
</tr>
<tr>
<td>\texttt{#\bel}</td>
<td>7</td>
</tr>
<tr>
<td>\texttt{#\bs}</td>
<td>8</td>
</tr>
<tr>
<td>\texttt{#\ht}</td>
<td>9</td>
</tr>
<tr>
<td>\texttt{#\lf}</td>
<td>10</td>
</tr>
<tr>
<td>\texttt{#\vt}</td>
<td>11</td>
</tr>
<tr>
<td>\texttt{#\ff}</td>
<td>12</td>
</tr>
<tr>
<td>\texttt{#\cr}</td>
<td>13</td>
</tr>
<tr>
<td>\texttt{#\so}</td>
<td>14</td>
</tr>
<tr>
<td>\texttt{#\si}</td>
<td>15</td>
</tr>
<tr>
<td>\texttt{#\dle}</td>
<td>16</td>
</tr>
<tr>
<td>\texttt{#\dc1}</td>
<td>17</td>
</tr>
<tr>
<td>\texttt{#\dc2}</td>
<td>18</td>
</tr>
<tr>
<td>\texttt{#\dc3}</td>
<td>19</td>
</tr>
<tr>
<td>\texttt{#\dc4}</td>
<td>20</td>
</tr>
<tr>
<td>\texttt{#\nak}</td>
<td>21</td>
</tr>
<tr>
<td>\texttt{#\syn}</td>
<td>22</td>
</tr>
<tr>
<td>\texttt{#\etb}</td>
<td>23</td>
</tr>
<tr>
<td>\texttt{#\can}</td>
<td>24</td>
</tr>
<tr>
<td>\texttt{#\em}</td>
<td>25</td>
</tr>
<tr>
<td>\texttt{#\sub}</td>
<td>26</td>
</tr>
<tr>
<td>\texttt{#\esc}</td>
<td>27</td>
</tr>
<tr>
<td>\texttt{#\fs}</td>
<td>28</td>
</tr>
<tr>
<td>\texttt{#\gs}</td>
<td>29</td>
</tr>
<tr>
<td>\texttt{#\rs}</td>
<td>30</td>
</tr>
<tr>
<td>\texttt{#\us}</td>
<td>31</td>
</tr>
<tr>
<td>\texttt{#\sp}</td>
<td>32</td>
</tr>
</tbody>
</table>

The short name for the “delete” character (code point U+007F) is \texttt{#\del}.

The R7RS name for the “escape” character (code point U+001B) is \texttt{#\escape}.

There are also a few alternative names left over for compatibility with previous versions of Guile.

<table>
<thead>
<tr>
<th>Alternate</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>#\nul</td>
<td>#\newline</td>
</tr>
<tr>
<td>#\np</td>
<td>#\page</td>
</tr>
</tbody>
</table>
Characters may also be written using their code point values. They can be written with an octal number, such as \10 for \bs or \177 for \del.

If one prefers hex to octal, there is an additional syntax for character escapes: \xHHHH – the letter 'x' followed by a hexadecimal number of one to eight digits.

\null

<table>
<thead>
<tr>
<th>Scheme Procedure</th>
<th>C Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>char? x</td>
<td>scm_char_p (x)</td>
</tr>
<tr>
<td>Return #t if x is a character, else #f.</td>
<td></td>
</tr>
</tbody>
</table>

Fundamentally, the character comparison operations below are numeric comparisons of the character’s code points.

<table>
<thead>
<tr>
<th>Scheme Procedure</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>char=? x y</td>
<td>Return #t if code point of x is equal to the code point of y, else #f.</td>
</tr>
<tr>
<td>char&lt;? x y</td>
<td>Return #t if the code point of x is less than the code point of y, else #f.</td>
</tr>
<tr>
<td>char&lt;=? x y</td>
<td>Return #t if the code point of x is less than or equal to the code point of y, else #f.</td>
</tr>
<tr>
<td>char&gt;? x y</td>
<td>Return #t if the code point of x is greater than the code point of y, else #f.</td>
</tr>
<tr>
<td>char&gt;=? x y</td>
<td>Return #t if the code point of x is greater than or equal to the code point of y, else #f.</td>
</tr>
</tbody>
</table>

Case-insensitive character comparisons use Unicode case folding. In case folding comparisons, if a character is lowercase and has an uppercase form that can be expressed as a single character, it is converted to uppercase before comparison. All other characters undergo no conversion before the comparison occurs. This includes the German sharp S (Eszett) which is not uppercased before conversion because its uppercase form has two characters. Unicode case folding is language independent: it uses rules that are generally true, but, it cannot cover all cases for all languages.

<table>
<thead>
<tr>
<th>Scheme Procedure</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>char-ci=? x y</td>
<td>Return #t if the case-folded code point of x is the same as the case-folded code point of y, else #f.</td>
</tr>
<tr>
<td>char-ci&lt;? x y</td>
<td>Return #t if the case-folded code point of x is less than the case-folded code point of y, else #f.</td>
</tr>
<tr>
<td>char-ci&lt;=? x y</td>
<td>Return #t if the case-folded code point of x is less than or equal to the case-folded code point of y, else #f.</td>
</tr>
<tr>
<td>char-ci&gt;? x y</td>
<td>Return #t if the case-folded code point of x is greater than the case-folded code point of y, else #f.</td>
</tr>
</tbody>
</table>
char-ci=>? x y
   [Scheme Procedure]
   Return #t if the case-folded code point of x is greater than or equal to the case-folded
code point of y, else #f.

cchar-alphabetic? chr
   [Scheme Procedure]
scm_char_alphabetic_p (chr)
   [C Function]
   Return #t if chr is alphabetic, else #f.

cchar-numeric? chr
   [Scheme Procedure]
scm_char_numeric_p (chr)
   [C Function]
   Return #t if chr is numeric, else #f.

cchar-whitespace? chr
   [Scheme Procedure]
scm_char_whitespace_p (chr)
   [C Function]
   Return #t if chr is whitespace, else #f.

cchar-upper-case? chr
   [Scheme Procedure]
scm_char_upper_case_p (chr)
   [C Function]
   Return #t if chr is uppercase, else #f.

cchar-lower-case? chr
   [Scheme Procedure]
scm_char_lower_case_p (chr)
   [C Function]
   Return #t if chr is lowercase, else #f.

cchar-is-both? chr
   [Scheme Procedure]
scm_char_is_both_p (chr)
   [C Function]
   Return #t if chr is either uppercase or lowercase, else #f.

cchar-general-category chr
   [Scheme Procedure]
scm_char_general_category (chr)
   [C Function]
   Return a symbol giving the two-letter name of the Unicode general category assigned
to chr or #f if no named category is assigned. The following table provides a list of
category names along with their meanings.

<table>
<thead>
<tr>
<th>Category</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lu</td>
<td>Uppercase letter</td>
</tr>
<tr>
<td>Li</td>
<td>Lowercase letter</td>
</tr>
<tr>
<td>Lt</td>
<td>Titlecase letter</td>
</tr>
<tr>
<td>Lm</td>
<td>Modifier letter</td>
</tr>
<tr>
<td>Lo</td>
<td>Other letter</td>
</tr>
<tr>
<td>Mn</td>
<td>Non-spacing mark</td>
</tr>
<tr>
<td>Mc</td>
<td>Combining spacing mark</td>
</tr>
<tr>
<td>Me</td>
<td>Enclosing mark</td>
</tr>
<tr>
<td>Nd</td>
<td>Decimal digit number</td>
</tr>
<tr>
<td>Nl</td>
<td>Letter number</td>
</tr>
<tr>
<td>No</td>
<td>Other number</td>
</tr>
<tr>
<td>Pc</td>
<td>Connector punctuation</td>
</tr>
<tr>
<td>Pd</td>
<td>Dash punctuation</td>
</tr>
<tr>
<td>Ps</td>
<td>Open punctuation</td>
</tr>
<tr>
<td>Pe</td>
<td>Close punctuation</td>
</tr>
<tr>
<td>Pi</td>
<td>Initial quote punctuation</td>
</tr>
<tr>
<td>Pf</td>
<td>Final quote punctuation</td>
</tr>
<tr>
<td>Po</td>
<td>Other punctuation</td>
</tr>
<tr>
<td>Sm</td>
<td>Math symbol</td>
</tr>
<tr>
<td>Sc</td>
<td>Currency symbol</td>
</tr>
<tr>
<td>Sk</td>
<td>Modifier symbol</td>
</tr>
<tr>
<td>So</td>
<td>Other symbol</td>
</tr>
<tr>
<td>Zs</td>
<td>Space separator</td>
</tr>
<tr>
<td>Zi</td>
<td>Line separator</td>
</tr>
<tr>
<td>Zp</td>
<td>Paragraph separator</td>
</tr>
<tr>
<td>Cc</td>
<td>Control</td>
</tr>
<tr>
<td>Cf</td>
<td>Format</td>
</tr>
<tr>
<td>Cs</td>
<td>Surrogate</td>
</tr>
<tr>
<td>Co</td>
<td>Private use</td>
</tr>
<tr>
<td>Cn</td>
<td>Unassigned</td>
</tr>
</tbody>
</table>
char->integer \textit{chr} \quad \text{[Scheme Procedure]}
\textbf{scm\_char\_to\_integer} (\textit{chr}) \quad \text{[C Function]}
Return the code point of \textit{chr}.

\textbf{integer->char} \textit{n} \quad \textbf{[Scheme Procedure]}
\textbf{scm\_integer\_to\_char} (\textit{n}) \quad \text{[C Function]}
Return the character that has code point \textit{n}. The integer \textit{n} must be a valid code point. Valid code points are in the ranges 0 to \#x7FF inclusive or \#xE000 to \#x10FFFF inclusive.

\textbf{char-upcase} \textit{chr} \quad \text{[Scheme Procedure]}
\textbf{scm\_char\_upcase} (\textit{chr}) \quad \text{[C Function]}
Return the uppercase character version of \textit{chr}.

\textbf{char-downcase} \textit{chr} \quad \text{[Scheme Procedure]}
\textbf{scm\_char\_downcase} (\textit{chr}) \quad \text{[C Function]}
Return the lowercase character version of \textit{chr}.

\textbf{char-titlecase} \textit{chr} \quad \text{[Scheme Procedure]}
\textbf{scm\_char\_titlecase} (\textit{chr}) \quad \text{[C Function]}
Return the titlecase character version of \textit{chr} if one exists; otherwise return the uppercase version.

For most characters these will be the same, but the Unicode Standard includes certain digraph compatibility characters, such as U+01F3 “dz”, for which the uppercase and titlecase characters are different (U+01F1 “DZ” and U+01F2 “Dz” in this case, respectively).

\textbf{scm\_t\_wchar} \textbf{scm\_c\_upcase} (\textbf{scm\_t\_wchar} \textit{c}) \quad \text{[C Function]}
\textbf{scm\_t\_wchar} \textbf{scm\_c\_downcase} (\textbf{scm\_t\_wchar} \textit{c}) \quad \text{[C Function]}
\textbf{scm\_t\_wchar} \textbf{scm\_c\_titlecase} (\textbf{scm\_t\_wchar} \textit{c}) \quad \text{[C Function]}
These C functions take an integer representation of a Unicode codepoint and return the codepoint corresponding to its uppercase, lowercase, and titlecase forms respectively. The type \textbf{scm\_t\_wchar} is a signed, 32-bit integer.

Characters also have “formal names”, which are defined by Unicode. These names can be accessed in Guile from the \textbf{(ice-9 unicode)} module:

\begin{verbatim}
(use-modules (ice-9 unicode))
\end{verbatim}

\textbf{char->formal-name} \textit{chr} \quad \text{[Scheme Procedure]}
Return the formal all-upper-case Unicode name of \textit{ch}, as a string, or \#f if the character has no name.

\textbf{formal-name->char} \textit{name} \quad \text{[Scheme Procedure]}
Return the character whose formal all-upper-case Unicode name is \textit{name}, or \#f if no such character is known.
6.6.4 Character Sets

The features described in this section correspond directly to SRFI-14.

The data type charset implements sets of characters (see Section 6.6.3 [Characters], page 129). Because the internal representation of character sets is not visible to the user, a lot of procedures for handling them are provided.

Character sets can be created, extended, tested for the membership of a character and be compared to other character sets.

6.6.4.1 Character Set Predicates/Comparison

Use these procedures for testing whether an object is a character set, or whether several character sets are equal or subsets of each other. char-set-hash can be used for calculating a hash value, maybe for usage in fast lookup procedures.

\[
\begin{align*}
\text{char-set? } \text{obj} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_p (obj)} & \quad \text{[C Function]} \\
\text{Return } \#t & \text{if } \text{obj} \text{ is a character set, } \#f \text{ otherwise.} \\
\text{char-set= char.set ...} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_eq (char.sets)} & \quad \text{[C Function]} \\
\text{Return } \#t & \text{if all given character sets are equal.} \\
\text{char-set<= char.set ...} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_leq (char.sets)} & \quad \text{[C Function]} \\
\text{Return } \#t & \text{if every character set } \text{char.set}_i \text{ is a subset of character set } \text{char.set}_{i+1}. \\
\text{char-set-hash cs [bound]} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_hash (cs, bound)} & \quad \text{[C Function]} \\
\text{Compute a hash value for the character set } \text{cs}. \text{ If } \text{bound} \text{ is given and non-zero, it restricts the returned value to the range } 0 \ldots \text{bound - 1}. \\
\end{align*}
\]

6.6.4.2 Iterating Over Character Sets

Character set cursors are a means for iterating over the members of a character set. After creating a character set cursor with char-set-cursor, a cursor can be dereferenced with char-set-ref, advanced to the next member with char-set-cursor-next. Whether a cursor has passed past the last element of the set can be checked with end-of-char-set?.

Additionally, mapping and (un-)folding procedures for character sets are provided.

\[
\begin{align*}
\text{char-set-cursor cs} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_cursor (cs)} & \quad \text{[C Function]} \\
\text{Return a cursor into the character set } \text{cs}. \\
\text{char-set-ref cs cursor} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_ref (cs, cursor)} & \quad \text{[C Function]} \\
\text{Return the character at the current cursor position } \text{cursor} \text{ in the character set } \text{cs}. \text{ It is an error to pass a cursor for which end-of-char-set? returns true.} \\
\text{char-set-cursor-next cs cursor} & \quad \text{[Scheme Procedure]} \\
\text{scm_char_set_cursor_next (cs, cursor)} & \quad \text{[C Function]} \\
\text{Advance the character set cursor } \text{cursor} \text{ to the next character in the character set } \text{cs}. \text{ It is an error if the cursor given satisfies end-of-char-set?}. \\
\end{align*}
\]
end-of-char-set? cursor  
scm_end_of_char_set_p (cursor)  
Return #t if cursor has reached the end of a character set, #f otherwise.

char-set-fold kons knil cs  
scm_char_set_fold (kons, knil, cs)  
Fold the procedure kons over the character set cs, initializing it with knil.

char-set-unfold p f g seed [base_cs]  
scm_char_set_unfold (p, f, g, seed, base_cs)  
This is a fundamental constructor for character sets.
  • g is used to generate a series of “seed” values from the initial seed: seed, (g seed), (g^2 seed), (g^3 seed), . . .
  • p tells us when to stop – when it returns true when applied to one of the seed values.
  • f maps each seed value to a character. These characters are added to the base character set base_cs to form the result; base_cs defaults to the empty set.

char-set-unfold! p f g seed base_cs  
scm_char_set_unfold_x (p, f, g, seed, base_cs)  
This is a fundamental constructor for character sets.
  • g is used to generate a series of “seed” values from the initial seed: seed, (g seed), (g^2 seed), (g^3 seed), . . .
  • p tells us when to stop – when it returns true when applied to one of the seed values.
  • f maps each seed value to a character. These characters are added to the base character set base_cs to form the result; base_cs defaults to the empty set.

char-set-for-each proc cs  
scm_char_set_for_each (proc, cs)  
Apply proc to every character in the character set cs. The return value is not specified.

char-set-map proc cs  
scm_char_set_map (proc, cs)  
Map the procedure proc over every character in cs. proc must be a character -> character procedure.

6.6.4.3 Creating Character Sets

New character sets are produced with these procedures.

char-set-copy cs  
scm_char_set_copy (cs)  
Return a newly allocated character set containing all characters in cs.

char-set chr . . .  
scm_char_set (chrs)  
Return a character set containing all given characters.
list->char-set  list [base_cs]  [Scheme Procedure]
scm_list_to_char_set (list, base_cs)  [C Function]
        Convert the character list list to a character set. If the character set base_cs is given, the character in this set are also included in the result.

list->char-set! list base_cs  [Scheme Procedure]
scm_list_to_char_set_x (list, base_cs)  [C Function]
        Convert the character list list to a character set. The characters are added to base_cs and base_cs is returned.

string->char-set  str [base_cs]  [Scheme Procedure]
scm_string_to_char_set (str, base_cs)  [C Function]
        Convert the string str to a character set. If the character set base_cs is given, the characters in this set are also included in the result.

string->char-set! str base_cs  [Scheme Procedure]
scm_string_to_char_set_x (str, base_cs)  [C Function]
        Convert the string str to a character set. The characters from the string are added to base_cs, and base_cs is returned.

char-set-filter pred cs [base_cs]  [Scheme Procedure]
scm_char_set_filter (pred, cs, base_cs)  [C Function]
        Return a character set containing every character from cs so that it satisfies pred. If provided, the characters from base_cs are added to the result.

char-set-filter! pred cs base_cs  [Scheme Procedure]
scm_char_set_filter_x (pred, cs, base_cs)  [C Function]
        Return a character set containing every character from cs so that it satisfies pred. The characters are added to base_cs and base_cs is returned.

ucs-range->char-set  lower upper [error [base_cs]]  [Scheme Procedure]
scm_ucs_range_to_char_set (lower, upper, error, base_cs)  [C Function]
        Return a character set containing all characters whose character codes lie in the half-open range [lower,upper).

        If error is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If error is #f, these characters are silently left out of the resulting character set.

        The characters in base_cs are added to the result, if given.

ucs-range->char-set! lower upper error base_cs  [Scheme Procedure]
scm_ucs_range_to_char_set_x (lower, upper, error, base_cs)  [C Function]
        Return a character set containing all characters whose character codes lie in the half-open range [lower,upper).

        If error is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If error is #f, these characters are silently left out of the resulting character set.

        The characters are added to base_cs and base_cs is returned.
Chapter 6: API Reference

->char-set x

scm_to_char_set (x)

Coerces x into a char-set. x may be a string, character or char-set. A string is converted to the set of its constituent characters; a character is converted to a singleton set; a char-set is returned as-is.

6.6.4.4 Querying Character Sets

Access the elements and other information of a character set with these procedures.

%char-set-dump cs

Returns an association list containing debugging information for cs. The association list has the following entries.

  char-set  The char-set itself
  len       The number of groups of contiguous code points the char-set contains
  ranges    A list of lists where each sublist is a range of code points and their associated characters

The return value of this function cannot be relied upon to be consistent between versions of Guile and should not be used in code.

char-set-size cs

scm_char_set_size (cs)

Return the number of elements in character set cs.

char-set-count pred cs

scm_char_set_count (pred, cs)

Return the number of the elements int the character set cs which satisfy the predicate pred.

char-set->list cs

scm_char_set_to_list (cs)

Return a list containing the elements of the character set cs.

char-set->string cs

scm_char_set_to_string (cs)

Return a string containing the elements of the character set cs. The order in which the characters are placed in the string is not defined.

char-set-contains? cs ch

scm_char_set_contains_p (cs, ch)

Return #t if the character ch is contained in the character set cs, or #f otherwise.

char-set-every pred cs

scm_char_set_every (pred, cs)

Return a true value if every character in the character set cs satisfies the predicate pred.
char-set-any pred cs

scm_char_set_any (pred, cs)

Return a true value if any character in the character set cs satisfies the predicate pred.

6.6.4.5 Character-Set Algebra

Character sets can be manipulated with the common set algebra operation, such as union, complement, intersection etc. All of these procedures provide side-effecting variants, which modify their character set argument(s).

char-set-adjoin cs chr ...  
scm_char_set_adjoin (cs, chrs)

Add all character arguments to the first argument, which must be a character set.

char-set-delete cs chr ...  
scm_char_set_delete (cs, chrs)

Delete all character arguments from the first argument, which must be a character set.

char-set-adjoin! cs chr ...  
scm_char_set_adjoin_x (cs, chrs)

Add all character arguments to the first argument, which must be a character set.

char-set-delete! cs chr ...  
scm_char_set_delete_x (cs, chrs)

Delete all character arguments from the first argument, which must be a character set.

char-set-complement cs  
scm_char_set_complement (cs)

Return the complement of the character set cs.

Note that the complement of a character set is likely to contain many reserved code points (code points that are not associated with characters). It may be helpful to modify the output of char-set-complement by computing its intersection with the set of designated code points, char-set:designated.

char-set-union cs ...  
scm_char_set_union (char_sets)

Return the union of all argument character sets.

char-set-intersection cs ...  
scm_char_set_intersection (char_sets)

Return the intersection of all argument character sets.

char-set-difference cs1 cs ...  
scm_char_set_difference (cs1, char_sets)

Return the difference of all argument character sets.
Chapter 6: API Reference

char-set-xor cs ...  

scm_char_set_xor (char_sets)  
Return the exclusive-or of all argument character sets.

char-set-diff+intersection cs1 cs ...  

scm_char_set_diff_plus_intersection (cs1, char_sets)  
Return the difference and the intersection of all argument character sets.

char-set-complement! cs  

scm_char_set_complement_x (cs)  
Return the complement of the character set cs.

char-set-union! cs1 cs ...  

scm_char_set_union_x (cs1, char_sets)  
Return the union of all argument character sets.

char-set-intersection! cs1 cs ...  

scm_char_set_intersection_x (cs1, char_sets)  
Return the intersection of all argument character sets.

char-set-difference! cs1 cs ...  

scm_char_set_difference_x (cs1, char_sets)  
Return the difference of all argument character sets.

char-set-xor! cs1 cs ...  

scm_char_set_xor_x (cs1, char_sets)  
Return the exclusive-or of all argument character sets.

char-set-diff+intersection! cs1 cs2 cs ...  

scm_char_set_diff_plus_intersection_x (cs1, cs2, char_sets)  
Return the difference and the intersection of all argument character sets.

6.6.4.6 Standard Character Sets

In order to make the use of the character set data type and procedures useful, several predefined character set variables exist.

These character sets are locale independent and are not recomputed upon a setlocale call. They contain characters from the whole range of Unicode code points. For instance, char-set:letter contains about 100,000 characters.

char-set:lower-case  

scm_char_set_lower_case  
All lower-case characters.

char-set:upper-case  

scm_char_set_upper_case  
All upper-case characters.

char-set:title-case  

scm_char_set_title_case  
All single characters that function as if they were an upper-case letter followed by a lower-case letter.
char-set:letter [Scheme Variable]
scm_char_set_letter [C Variable]
   All letters. This includes char-set:lower-case, char-set:upper-case,
   char-set:title-case, and many letters that have no case at all. For example,
   Chinese and Japanese characters typically have no concept of case.

char-set:digit [Scheme Variable]
scm_char_set_digit [C Variable]
   All digits.

char-set:letter+digit [Scheme Variable]
scm_char_set_letter_and_digit [C Variable]

char-set:graphic [Scheme Variable]
scm_char_set_graphic [C Variable]
   All characters which would put ink on the paper.

char-set:printing [Scheme Variable]
scm_char_set_printing [C Variable]
   The union of char-set:graphic and char-set:whitespace.

char-set:whitespace [Scheme Variable]
scm_char_set_whitespace [C Variable]
   All whitespace characters.

char-set:blank [Scheme Variable]
scm_char_set_blank [C Variable]
   All horizontal whitespace characters, which notably includes \\space and \\tab.

char-set:iso-control [Scheme Variable]
scm_char_set_iso_control [C Variable]
   The ISO control characters are the C0 control characters (U+0000 to U+001F), delete
   (U+007F), and the C1 control characters (U+0080 to U+009F).

char-set:punctuation [Scheme Variable]
scm_char_set_punctuation [C Variable]
   All punctuation characters, such as the characters !"#$%&'()*+,-./:;<=>?@[\]^_`{|}.

char-set:symbol [Scheme Variable]
scm_char_set_symbol [C Variable]
   All symbol characters, such as the characters $+_<=>`~`\.

char-set:hex-digit [Scheme Variable]
scm_char_set_hex_digit [C Variable]
   The hexadecimal digits 0123456789abcdefABCDEF.

char-set:ascii [Scheme Variable]
scm_char_set_ascii [C Variable]
   All ASCII characters.
### 6.6.5 Strings

Strings are fixed-length sequences of characters. They can be created by calling constructor procedures, but they can also literally get entered at the REPL or in Scheme source files.

Strings always carry the information about how many characters they are composed of with them, so there is no special end-of-string character, like in C. That means that Scheme strings can contain any character, even the ‘\nul’ character ‘\0’.

To use strings efficiently, you need to know a bit about how Guile implements them. In Guile, a string consists of two parts, a head and the actual memory where the characters are stored. When a string (or a substring of it) is copied, only a new head gets created, the memory is usually not copied. The two heads start out pointing to the same memory.

When one of these two strings is modified, as with string-set!, their common memory does get copied so that each string has its own memory and modifying one does not accidentally modify the other as well. Thus, Guile’s strings are ‘copy on write’; the actual copying of their memory is delayed until one string is written to.

This implementation makes functions like substring very efficient in the common case that no modifications are done to the involved strings.

If you do know that your strings are getting modified right away, you can use substring/copy instead of substring. This function performs the copy immediately at the time of creation. This is more efficient, especially in a multi-threaded program. Also, substring/copy can avoid the problem that a short substring holds on to the memory of a very large original string that could otherwise be recycled.

If you want to avoid the copy altogether, so that modifications of one string show up in the other, you can use substring/shared. The strings created by this procedure are called mutation sharing substrings since the substring and the original string share modifications to each other.

If you want to prevent modifications, use substring/read-only.

Guile provides all procedures of SRFI-13 and a few more.

#### 6.6.5.1 String Read Syntax

The read syntax for strings is an arbitrarily long sequence of characters enclosed in double quotes (").
Backslash is an escape character and can be used to insert the following special characters. \" and \ are R5RS standard, \| is R7RS standard, the next seven are R6RS standard — notice they follow C syntax — and the remaining four are Guile extensions.

- Backslash character.
- Double quote character (an unescaped " is otherwise the end of the string).
- Vertical bar character.
- Bell character (ASCII 7).
- Formfeed character (ASCII 12).
- Newline character (ASCII 10).
- Carriage return character (ASCII 13).
- Tab character (ASCII 9).
- Vertical tab character (ASCII 11).
- Backspace character (ASCII 8).
- NUL character (ASCII 0).
- Open parenthesis. This is intended for use at the beginning of lines in multiline strings to avoid confusing Emacs lisp modes.

\ followed by newline (ASCII 10)
Nothing. This way if \ is the last character in a line, the string will continue with the first character from the next line, without a line break.
If the hungry-eol-escapes reader option is enabled, which is not the case by default, leading whitespace on the next line is discarded.

```
"foo\nbar"
⇒ "foo  bar"
(read-enable 'hungry-eol-escapes)
"foo\nbar"
⇒ "foobar"
```

- Character code given by two hexadecimal digits. For example \x7f for an ASCII DEL (127).
- Character code given by four hexadecimal digits. For example \u0100 for a capital A with macron (U+0100).
- Character code given by six hexadecimal digits. For example \U010402.

The following are examples of string literals:

```
"foo"
"bar plonk"
"Hello World"
"\"Hi\", he said."
```
The three escape sequences \xHH, \uHHHH and \UHHHHHH were chosen to not break compatibility with code written for previous versions of Guile. The R6RS specification suggests a different, incompatible syntax for hex escapes: \xHHHH; – a character code followed by one to eight hexadecimal digits terminated with a semicolon. If this escape format is desired instead, it can be enabled with the reader option \r6rs-hex-escapes.

(read-enable 'r6rs-hex-escapes)

For more on reader options, See Section 6.18.2 [Scheme Read], page 382.

### 6.6.5.2 String Predicates

The following procedures can be used to check whether a given string fulfills some specified property.

**string? obj**  
**[Scheme Procedure]**

**scm_string_p (obj)**  
**[C Function]**

Return \#t if \obj is a string, else \#f.

**int scm_is_string (SCM obj)**  
**[C Function]**

Returns 1 if \obj is a string, 0 otherwise.

**string-null? str**  
**[Scheme Procedure]**

**scm_string_null_p (str)**  
**[C Function]**

Return \#t if \str's length is zero, and \#f otherwise.

```scheme
(string-null? "") ⇒ \#t  
y ⇒ "foo"  
(string-null? y) ⇒ \#f
```

**string-any char_pred s [start [end]]**  
**[Scheme Procedure]**

**scm_string_any (char_pred, s, start, end)**  
**[C Function]**

Check if \char_pred is true for any character in string \s.

\char_pred can be a character to check for any equal to that, or a character set (see Section 6.6.4 [Character Sets], page 134) to check for any in that set, or a predicate procedure to call.

For a procedure, calls \(\langle \text{char_pred}\ c\rangle\) are made successively on the characters from \start to \end. If \char_pred returns true (ie. non-\#f), \string-any stops and that return value is the return from \string-any. The call on the last character (ie. at \end − 1), if that point is reached, is a tail call.

If there are no characters in \s (ie. \start equals \end) then the return is \#f.

**string-every char_pred s [start [end]]**  
**[Scheme Procedure]**

**scm_string_every (char_pred, s, start, end)**  
**[C Function]**

Check if \char_pred is true for every character in string \s.

\char_pred can be a character to check for every character equal to that, or a character set (see Section 6.6.4 [Character Sets], page 134) to check for every character being in that set, or a predicate procedure to call.

For a procedure, calls \(\langle \text{char_pred}\ c\rangle\) are made successively on the characters from \start to \end. If \char_pred returns \#f, \string-every stops and returns \#f. The call
on the last character (ie. at \textit{end} − 1), if that point is reached, is a tail call and the
return from that call is the return from \texttt{string-every}.
If there are no characters in \textit{s} (ie. \textit{start} equals \textit{end}) then the return is \texttt{#t}.

6.6.5.3 String Constructors

The string constructor procedures create new string objects, possibly initializing them with
some specified character data. See also See Section 6.6.5.5 [String Selection], page 145, for
ways to create strings from existing strings.

\texttt{string char...} \hspace{1cm} \texttt{[Scheme Procedure]}
\begin{itemize}
  \item \texttt{(string #\textbackslash x #\textbackslash y #\textbackslash z)} \Rightarrow "xyz"
  \item \texttt{(string)} \Rightarrow ""
\end{itemize}

\texttt{list->string lst} \hspace{1cm} \texttt{[Scheme Procedure]}
scm\_string (lst) \hspace{1cm} \texttt{[C Function]}
\begin{itemize}
  \item \texttt{(list->string '(#\textbackslash a #\textbackslash b #\textbackslash c)) \Rightarrow "abc"}
\end{itemize}

\texttt{reverse-list->string lst} \hspace{1cm} \texttt{[Scheme Procedure]}
scm\_reverse_list_to_string (lst) \hspace{1cm} \texttt{[C Function]}
\begin{itemize}
  \item \texttt{(reverse-list->string '(#\textbackslash a #\textbackslash B #\textbackslash c)) \Rightarrow "cBa"}
\end{itemize}

\texttt{make-string k [chr]} \hspace{1cm} \texttt{[Scheme Procedure]}
scm\_make_string (k, chr) \hspace{1cm} \texttt{[C Function]}
\begin{itemize}
  \item Return a newly allocated string of length \textit{k}. If \textit{chr} is given, then all elements of the
  string are initialized to \textit{chr}, otherwise the contents of the string are unspecified.
\end{itemize}

\texttt{SCM scm\_c\_make\_string (size\_t len, SCM chr)} \hspace{1cm} \texttt{[C Function]}
\begin{itemize}
  \item Like \texttt{scm\_make\_string}, but expects the length as a \texttt{size\_t}.
\end{itemize}

\texttt{string-tabulate proc len} \hspace{1cm} \texttt{[Scheme Procedure]}
scm\_string\_tabulate (proc, len) \hspace{1cm} \texttt{[C Function]}
\begin{itemize}
  \item \texttt{proc} is an integer->char procedure. Construct a string of size \textit{len} by applying \texttt{proc} to
each index to produce the corresponding string element. The order in which \texttt{proc} is
applied to the indices is not specified.
\end{itemize}

\texttt{string-join ls [delimiter [grammar]]} \hspace{1cm} \texttt{[Scheme Procedure]}
scm\_string\_join (ls, delimiter, grammar) \hspace{1cm} \texttt{[C Function]}
\begin{itemize}
  \item Append the string in the string list \textit{ls}, using the string \texttt{delimiter} as a delimiter between
the elements of \textit{ls}. \texttt{grammar} is a symbol which specifies how the delimiter is placed
between the strings, and defaults to the symbol \texttt{infix}.
\end{itemize}

\texttt{infix} \hspace{1cm} Insert the separator between list elements. An empty string will produce
an empty list.

\texttt{strict-infix} \hspace{1cm} Like \texttt{infix}, but will raise an error if given the empty list.
6.6.5.4 List/String conversion

When processing strings, it is often convenient to first convert them into a list representation by using the procedure `string->list`, work with the resulting list, and then convert it back into a string. These procedures are useful for similar tasks.

**Scheme Procedure**

```
string->list str [start [end]]
```

Convert the string `str` into a list of characters.

**C Function**

```
scm_string_to_list (str)
```

Convert the string `str` into a list of characters.

**Scheme Procedure**

```
string-split str char_pred
```

Split the string `str` into a list of substrings delimited by appearances of characters that

- equal `char_pred`, if it is a character,
- satisfy the predicate `char_pred`, if it is a procedure,
- are in the set `char_pred`, if it is a character set.

Note that an empty substring between separator characters will result in an empty string in the result list.

```
(string-split "root:x:0:0:root:/root:/bin/bash" #\:)
⇒
("root" "x" "0" "0" "root" "/root" "/bin/bash")
```

```
(string-split "::" #\:)
⇒
("" "::"
```

```
(string-split "" #\:)
⇒
(""
```

6.6.5.5 String Selection

Portions of strings can be extracted by these procedures. `string-ref` delivers individual characters whereas `substring` can be used to extract substrings from longer strings.

**Scheme Procedure**

```
string-length string
```

Return the number of characters in `string`.

**C Function**

```
size_t scm_c_string_length (SCM str)
```

Return the number of characters in `str` as a `size_t`.

**Scheme Procedure**

```
string-ref str k
```

Return character `k` of `str` using zero-origin indexing. `k` must be a valid index of `str`.
SCM scm_c_string_ref (SCM str, size_t k)    [C Function]
Return character k of str using zero-origin indexing. k must be a valid index of str.

string-copy str [start [end]]    [Scheme Procedure]
scm_substring_copy (str, start, end)    [C Function]
scm_string_copy (str)    [C Function]
Return a copy of the given string str.
The returned string shares storage with str initially, but it is copied as soon as one of the two strings is modified.

substring str start [end]    [Scheme Procedure]
scm_substring (str, start, end)    [C Function]
Return a new string formed from the characters of str beginning with index start (inclusive) and ending with index end (exclusive). str must be a string, start and end must be exact integers satisfying:
0 <= start <= end <= (string-length str).
The returned string shares storage with str initially, but it is copied as soon as one of the two strings is modified.

substring/shared str start [end]    [Scheme Procedure]
scm_substring_shared (str, start, end)    [C Function]
Like substring, but the strings continue to share their storage even if they are modified. Thus, modifications to str show up in the new string, and vice versa.

substring/copy str start [end]    [Scheme Procedure]
scm_substring_copy (str, start, end)    [C Function]
Like substring, but the storage for the new string is copied immediately.

substring/read-only str start [end]    [Scheme Procedure]
scm_substring_read_only (str, start, end)    [C Function]
Like substring, but the resulting string can not be modified.

SCM scm_c_substring (SCM str, size_t start, size_t end)    [C Function]
SCM scm_c_substring_shared (SCM str, size_t start, size_t end)    [C Function]
SCM scm_c_substring_copy (SCM str, size_t start, size_t end)    [C Function]
SCM scm_c_substring_read_only (SCM str, size_t start, size_t end)    [C Function]
Like scm_substring, etc. but the bounds are given as a size_t.

string-take s n    [Scheme Procedure]
scm_string_take (s, n)    [C Function]
Return the n first characters of s.

string-drop s n    [Scheme Procedure]
scm_string_drop (s, n)    [C Function]
Return all but the first n characters of s.

string-take-right s n    [Scheme Procedure]
scm_string_take_right (s, n)    [C Function]
Return the n last characters of s.
string-drop-right \textit{s n} \hspace{1cm} \textbf{[Scheme Procedure]}

\texttt{scm_string_drop_right (s, n)} \hspace{1cm} \textbf{[C Function]}

Return all but the last \textit{n} characters of \textit{s}.

\texttt{string-pad \textit{s len [chr [start [end]]]} \hspace{1cm} \textbf{[Scheme Procedure]}}

\texttt{string-pad-right \textit{s len [chr [start [end]]]} \hspace{1cm} \textbf{[Scheme Procedure]}}

\texttt{scm_string_pad (s, len, chr, start, end)} \hspace{1cm} \textbf{[C Function]}

\texttt{scm_string_pad_right (s, len, chr, start, end)} \hspace{1cm} \textbf{[C Function]}

Take characters \textit{start} to \textit{end} from the string \textit{s} and either pad with \textit{chr} or truncate them to give \textit{len} characters.

\texttt{string-pad} pads or truncates on the left, so for example

- \texttt{(string-pad "x" 3)} ⇒ "x"
- \texttt{(string-pad "abcde" 3)} ⇒ "cde"

\texttt{string-pad-right} pads or truncates on the right, so for example

- \texttt{(string-pad-right "x" 3)} ⇒ "x "
- \texttt{(string-pad-right "abcde" 3)} ⇒ "abc"

\texttt{string-trim \textit{s [char_pred [start [end]]]} \hspace{1cm} \textbf{[Scheme Procedure]}}

\texttt{string-trim-right \textit{s [char_pred [start [end]]]} \hspace{1cm} \textbf{[Scheme Procedure]}}

\texttt{string-trim-both \textit{s [char_pred [start [end]]]} \hspace{1cm} \textbf{[Scheme Procedure]}}

\texttt{scm_string_trim (s, char_pred, start, end)} \hspace{1cm} \textbf{[C Function]}

\texttt{scm_string_trim_right (s, char_pred, start, end)} \hspace{1cm} \textbf{[C Function]}

\texttt{scm_string_trim_both (s, char_pred, start, end)} \hspace{1cm} \textbf{[C Function]}

Trim occurrences of \textit{char_pred} from the ends of \textit{s}.

\texttt{string-trim} trims \textit{char_pred} characters from the left (\textit{start}) of the string,
\texttt{string-trim-right} trims them from the right (\textit{end}) of the string, \texttt{string-trim-both} trims from both ends.

\textit{char_pred} can be a character, a character set, or a predicate procedure to call on each character. If \textit{char_pred} is not given the default is whitespace as per \texttt{char-set:whitespace} (see Section 6.6.4.6 [Standard Character Sets], page 139).

- \texttt{(string-trim " x ")} ⇒ "x"
- \texttt{(string-trim-right "banana" #\a)} ⇒ "banan"
- \texttt{(string-trim-both ".", "xy:;" char-set:punctuation)} ⇒ "xy"
- \texttt{(string-trim-both "xyzzy" (lambda (c)
  (or (eqv? c #\x)
    (eqv? c #\y)))))} ⇒ "zz"

### 6.6.5.6 String Modification

These procedures are for modifying strings in-place. This means that the result of the operation is not a new string; instead, the original string’s memory representation is modified.

\texttt{string-set! \textit{str k chr} \hspace{1cm} \textbf{[Scheme Procedure]}}

\texttt{scm_string_set_x (str, k, chr)} \hspace{1cm} \textbf{[C Function]}

Store \textit{chr} in element \textit{k} of \textit{str} and return an unspecified value. \textit{k} must be a valid index of \textit{str}. 
void scm_c_string_set_x (SCM str, size_t k, SCM chr)  [C Function]

Like scm_string_set_x, but the index is given as a size_t.

string-fill! str chr [start [end]]  [Scheme Procedure]
scm_substring_fill_x (str, chr, start, end)  [C Function]
scm_string_fill_x (str, chr)  [C Function]

Stores chr in every element of the given str and returns an unspecified value.

substring-fill! str start end fill  [Scheme Procedure]
scm_substring_fill_x (str, start, end, fill)  [C Function]

Change every character in str between start and end to fill.

(define y (string-copy "abcdefg"))
(substring-fill! y 1 3 #\r)
y
⇒ "arrdefg"

substring-move! str1 start1 end1 str2 start2  [Scheme Procedure]
scm_substring_move_x (str1, start1, end1, str2, start2)  [C Function]

Copy the substring of str1 bounded by start1 and end1 into str2 beginning at position start2. str1 and str2 can be the same string.

string-copy! target tstart s [start [end]]  [Scheme Procedure]
scm_string_copy_x (target, tstart, s, start, end)  [C Function]

Copy the sequence of characters from index range [start, end] in string s to string target, beginning at index tstart. The characters are copied left-to-right or right-to-left as needed – the copy is guaranteed to work, even if target and s are the same string. It is an error if the copy operation runs off the end of the target string.

6.6.5.7 String Comparison

The procedures in this section are similar to the character ordering predicates (see Section 6.6.3 [Characters], page 129), but are defined on character sequences.

The first set is specified in R5RS and has names that end in ?. The second set is specified in SRFI-13 and the names have not ending ?.

The predicates ending in -ci ignore the character case when comparing strings. For now, case-insensitive comparison is done using the R5RS rules, where every lower-case character that has a single character upper-case form is converted to uppercase before comparison. See See Section 6.25.2 [Text Collation], page 460, for locale-dependent string comparison.

string=? s1 s2 s3 . . .  [Scheme Procedure]

Lexicographic equality predicate; return #t if all strings are the same length and contain the same characters in the same positions, otherwise return #f.

The procedure string-ci=? treats upper and lower case letters as though they were the same character, but string=? treats upper and lower case as distinct characters.

string<? s1 s2 s3 . . .  [Scheme Procedure]

Lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments str_i and str_i+1, str_i is lexicographically less than str_i+1.
**string<=?** s1 s2 s3 . . .  
Lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically less than or equal to \(str_{i+1}\).

**string>?** s1 s2 s3 . . .  
Lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically greater than \(str_{i+1}\).

**string>=?** s1 s2 s3 . . .  
Lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically greater than or equal to \(str_{i+1}\).

**string-ci=?** s1 s2 s3 . . .  
Case-insensitive string equality predicate; return #t if all strings are the same length and their component characters match (ignoring case) at each position; otherwise return #f.

**string-ci<?** s1 s2 s3 . . .  
Case-insensitive lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically less than \(str_{i+1}\) regardless of case.

**string-ci<=?** s1 s2 s3 . . .  
Case insensitive lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically less than or equal to \(str_{i+1}\) regardless of case.

**string-ci>?** s1 s2 s3 . . .  
Case insensitive lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically greater than \(str_{i+1}\) regardless of case.

**string-ci>=?** s1 s2 s3 . . .  
Case insensitive lexicographic ordering predicate; return #t if, for every pair of consecutive string arguments \(str_i\) and \(str_{i+1}\), \(str_i\) is lexicographically greater than or equal to \(str_{i+1}\) regardless of case.

**string-compare** s1 s2 proc_lt proc_eq proc_gt [start1 [end1] [start2 [end2]]]  
[Scheme Procedure]

**scm_string_compare** (s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2)  
[C Function]

Apply proc_lt, proc_eq, proc_gt to the mismatch index, depending upon whether s1 is less than, equal to, or greater than s2. The mismatch index is the largest index \(i\) such that for every \(0 \leq j < i\), \(s1[j] = s2[j]\) – that is, \(i\) is the first position that does not match.
string-compare-ci s1 s2 proc_lt proc_eq proc_gt [start1 [end1 [start2 [end2]]]]

Apply proc_lt, proc_eq, proc_gt to the mismatch index, depending upon whether s1 is less than, equal to, or greater than s2. The mismatch index is the largest index i such that for every 0 <= j < i, s1[j] = s2[j] – that is, i is the first position where the lowercased letters do not match.

string= s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 and s2 are not equal, a true value otherwise.

string<> s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 and s2 are equal, a true value otherwise.

string< s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 is greater or equal to s2, a true value otherwise.

string> s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 is less or equal to s2, a true value otherwise.

string<= s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 is greater to s2, a true value otherwise.

string>= s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 is less to s2, a true value otherwise.

string-ci= s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 and s2 are not equal, a true value otherwise. The character comparison is done case-insensitively.

string-ci<> s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 and s2 are equal, a true value otherwise. The character comparison is done case-insensitively.

string-ci< s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 is greater or equal to s2, a true value otherwise. The character comparison is done case-insensitively.

string-ci> s1 s2 [start1 [end1 [start2 [end2]]]]

Return #f if s1 is less or equal to s2, a true value otherwise. The character comparison is done case-insensitively.
Chapter 6: API Reference

string-ci<= s1 s2 [start1 [end1 [start2 [end2]]]]

scm_string_ci_le (s1, s2, start1, end1, start2, end2)

Return #f if s1 is greater to s2, a true value otherwise. The character comparison is done case-insensitively.

string-ci>= s1 s2 [start1 [end1 [start2 [end2]]]]

scm_string_ci_ge (s1, s2, start1, end1, start2, end2)

Return #f if s1 is less to s2, a true value otherwise. The character comparison is done case-insensitively.

string-hash s [bound [start [end]]]

scm_substring_hash (s, bound, start, end)

Compute a hash value for s. The optional argument bound is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

string-hash-ci s [bound [start [end]]]

scm_substring_hash_ci (s, bound, start, end)

Compute a hash value for s. The optional argument bound is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

Because the same visual appearance of an abstract Unicode character can be obtained via multiple sequences of Unicode characters, even the case-insensitive string comparison functions described above may return #f when presented with strings containing different representations of the same character. For example, the Unicode character “LATIN SMALL LETTER S WITH DOT BELOW AND DOT ABOVE” can be represented with a single character (U+1E69) or by the character “LATIN SMALL LETTER S” (U+0073) followed by the combining marks “COMBINING DOT BELOW” (U+0323) and “COMBINING DOT ABOVE” (U+0307).

For this reason, it is often desirable to ensure that the strings to be compared are using a mutually consistent representation for every character. The Unicode standard defines two methods of normalizing the contents of strings: Decomposition, which breaks composite characters into a set of constituent characters with an ordering defined by the Unicode Standard; and composition, which performs the converse.

There are two decomposition operations. “Canonical decomposition” produces character sequences that share the same visual appearance as the original characters, while “compatibility decomposition” produces ones whose visual appearances may differ from the originals but which represent the same abstract character.

These operations are encapsulated in the following set of normalization forms:

NFD Characters are decomposed to their canonical forms.

NFKD Characters are decomposed to their compatibility forms.

NFC Characters are decomposed to their canonical forms, then composed.

NFKC Characters are decomposed to their compatibility forms, then composed.

The functions below put their arguments into one of the forms described above.
\textbf{string-normalize-nfd} \ s \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_normalize\_nfd} \ (s) \quad \text{[C Function]}
\begin{itemize}
  \item Return the NFD normalized form of \textit{s}.
\end{itemize}

\textbf{string-normalize-nfkd} \ s \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_normalize\_nfkd} \ (s) \quad \text{[C Function]}
\begin{itemize}
  \item Return the NFKD normalized form of \textit{s}.
\end{itemize}

\textbf{string-normalize-nfc} \ s \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_normalize\_nfc} \ (s) \quad \text{[C Function]}
\begin{itemize}
  \item Return the NFC normalized form of \textit{s}.
\end{itemize}

\textbf{string-normalize-nfkc} \ s \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_normalize\_nfkc} \ (s) \quad \text{[C Function]}
\begin{itemize}
  \item Return the NFKC normalized form of \textit{s}.
\end{itemize}

\section*{6.6.5.8 String Searching}

\textbf{string-index} \ s \ char\_pred \ [start \ [end]] \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_index} \ (s, \ char\_pred, \ start, \ end) \quad \text{[C Function]}
\begin{itemize}
  \item Search through the string \textit{s} from left to right, returning the index of the first occurrence of a character which
  \begin{itemize}
    \item equals \textit{char\_pred}, if it is character,
    \item satisfies the predicate \textit{char\_pred}, if it is a procedure,
    \item is in the set \textit{char\_pred}, if it is a character set.
  \end{itemize}
\end{itemize}
\begin{itemize}
  \item Return \texttt{#f} if no match is found.
\end{itemize}

\textbf{string-rindex} \ s \ char\_pred \ [start \ [end]] \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_rindex} \ (s, \ char\_pred, \ start, \ end) \quad \text{[C Function]}
\begin{itemize}
  \item Search through the string \textit{s} from right to left, returning the index of the last occurrence of a character which
  \begin{itemize}
    \item equals \textit{char\_pred}, if it is character,
    \item satisfies the predicate \textit{char\_pred}, if it is a procedure,
    \item is in the set if \textit{char\_pred} is a character set.
  \end{itemize}
\end{itemize}
\begin{itemize}
  \item Return \texttt{#f} if no match is found.
\end{itemize}

\textbf{string-prefix-length} \ s1 \ s2 \ [start1 \ [end1] \ [start2 \ [end2]]] \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_prefix\_length} \ (s1, \ s2, \ start1, \ end1, \ start2, \ end2) \quad \text{[C Function]}
\begin{itemize}
  \item Return the length of the longest common prefix of the two strings.
\end{itemize}

\textbf{string-prefix-length-ci} \ s1 \ s2 \ [start1 \ [end1] \ [start2 \ [end2]]] \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_prefix\_length\_ci} \ (s1, \ s2, \ start1, \ end1, \ start2, \ end2) \quad \text{[C Function]}
\begin{itemize}
  \item Return the length of the longest common prefix of the two strings, ignoring character case.
\end{itemize}

\textbf{string-suffix-length} \ s1 \ s2 \ [start1 \ [end1] \ [start2 \ [end2]]] \quad \text{[Scheme Procedure]}
\texttt{scm\_string\_suffix\_length} \ (s1, \ s2, \ start1, \ end1, \ start2, \ end2) \quad \text{[C Function]}
\begin{itemize}
  \item Return the length of the longest common suffix of the two strings.
\end{itemize}
string-suffix-length-ci s1 s2 [start1 [end1 [start2 [end2]]]]

Return the length of the longest common suffix of the two strings, ignoring character case.

string-prefix? s1 s2 [start1 [end1 [start2 [end2]]]]

Is s1 a prefix of s2?

string-prefix-ci? s1 s2 [start1 [end1 [start2 [end2]]]]

Is s1 a prefix of s2, ignoring character case?

string-suffix? s1 s2 [start1 [end1 [start2 [end2]]]]

Is s1 a suffix of s2?

string-suffix-ci? s1 s2 [start1 [end1 [start2 [end2]]]]

Is s1 a suffix of s2, ignoring character case?

string-index-right s char_pred [start [end]]

Search through the string s from right to left, returning the index of the last occurrence of a character which
• equals char_pred, if it is character,
• satisfies the predicate char_pred, if it is a procedure,
• is in the set if char_pred is a character set.

string-skip s char_pred [start [end]]

Search through the string s from left to right, returning the index of the first occurrence of a character which
• does not equal char_pred, if it is character,
• does not satisfy the predicate char_pred, if it is a procedure,
• is not in the set if char_pred is a character set.

string-skip-right s char_pred [start [end]]

Search through the string s from right to left, returning the index of the last occurrence of a character which
• does not equal char_pred, if it is character,
• does not satisfy the predicate char_pred, if it is a procedure,
• is not in the set if char_pred is a character set.
**string-count**  
`s char_pred [start [end]]`  

[Scheme Procedure]  

Return the count of the number of characters in the string `s` which

- equals `char_pred`, if it is character,
- satisfies the predicate `char_pred`, if it is a procedure.
- is in the set `char_pred`, if it is a character set.

**string-contains**  
`s1 s2 [start1 [end1] [start2 [end2]]]`  

[Scheme Procedure]  

Does string `s1` contain string `s2`? Return the index in `s1` where `s2` occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings.

**string-contains-ci**  
`s1 s2 [start1 [end1] [start2 [end2]]]`  

[Scheme Procedure]  

Does string `s1` contain string `s2`? Return the index in `s1` where `s2` occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings. Character comparison is done case-insensitively.

### 6.6.5.9 Alphabetic Case Mapping

These are procedures for mapping strings to their upper- or lower-case equivalents, respectively, or for capitalizing strings.

They use the basic case mapping rules for Unicode characters. No special language or context rules are considered. The resulting strings are guaranteed to be the same length as the input strings.

See Section 6.25.3 [Character Case Mapping], page 461, for locale-dependent case conversions.

**string-upcase**  
`str [start [end]]`  

[Scheme Procedure]  

Upcase every character in `str`.

**string-upcase!**  
`str [start [end]]`  

[Scheme Procedure]  

Destructively upcase every character in `str`.

```
(string-upcase! y)
⇒ "ARRDEFG"
y
⇒ "ARRDEFG"
```

**string-downcase**  
`str [start [end]]`  

[Scheme Procedure]  

Downcase every character in `str`.  

**C Function**

---

---
string-downcase! str [start [end]]  [Scheme Procedure]
scm_substring_downcase_x (str, start, end)  [C Function]
scm_string_downcase_x (str)  [C Function]
  Destructively downcase every character in str.
  
y
  ⇒ "ARRDEFG"
  (string-downcase! y)
  ⇒ "arrdefg"
y
  ⇒ "arrdefg"

string-capitalize str  [Scheme Procedure]
scm_string_capitalize (str)  [C Function]
  Return a freshly allocated string with the characters in str, where the first character
  of every word is capitalized.

string-capitalize! str  [Scheme Procedure]
scm_string_capitalize_x (str)  [C Function]
  Upcase the first character of every word in str destructively and return str.
  
y
  ⇒ "hello world"
  (string-capitalize! y)
  ⇒ "Hello World"
y
  ⇒ "Hello World"

string-titlecase str [start [end]]  [Scheme Procedure]
scm_string_titlecase (str, start, end)  [C Function]
  Titlecase every first character in a word in str.

string-titlecase! str [start [end]]  [Scheme Procedure]
scm_string_titlecase_x (str, start, end)  [C Function]
  Destructively titlecase every first character in a word in str.

6.6.5.10 Reversing and Appending Strings

string-reverse str [start [end]]  [Scheme Procedure]
scm_string_reverse (str, start, end)  [C Function]
  Reverse the string str. The optional arguments start and end delimit the region of
  str to operate on.

string-reverse! str [start [end]]  [Scheme Procedure]
scm_string_reverse_x (str, start, end)  [C Function]
  Reverse the string str in-place. The optional arguments start and end delimit the
  region of str to operate on. The return value is unspecified.

string-append arg . . .  [Scheme Procedure]
scm_string_append (args)  [C Function]
  Return a newly allocated string whose characters form the concatenation of the given
  strings, arg . . .
  
  (let ((h "hello "))
    (string-append h "world"))
  ⇒ "hello world"
string-append/shared arg . . . [Scheme Procedure]
scm_string_append_shared (args) [C Function]
Like string-append, but the result may share memory with the argument strings.

string-concatenate ls [Scheme Procedure]
scm_string_concatenate (ls) [C Function]
Append the elements (which must be strings) of ls together into a single string.
Guaranteed to return a freshly allocated string.

string-concatenate-reverse ls [final_string end] [Scheme Procedure]
scm_string_concatenate_reverse (ls, final_string, end) [C Function]
Without optional arguments, this procedure is equivalent to

(string-concatenate (reverse ls))

If the optional argument final_string is specified, it is consed onto the beginning to ls
before performing the list-reverse and string-concatenate operations. If end is given,
only the characters of final_string up to index end are used.
Guaranteed to return a freshly allocated string.

string-concatenate/shared ls [Scheme Procedure]
scm_string_concatenate_shared (ls) [C Function]
Like string-concatenate, but the result may share memory with the strings in the
list ls.

string-concatenate-reverse/shared ls [final_string end] [Scheme Procedure]
scm_string_concatenate_reverse_shared (ls, final_string, end) [C Function]
Like string-concatenate-reverse, but the result may share memory with the
strings in the ls arguments.

6.6.5.11 Mapping, Folding, and Unfolding

string-map proc s [start [end]] [Scheme Procedure]
scm_string_map (proc, s, start, end) [C Function]
proc is a char->char procedure, it is mapped over s. The order in which the procedure
is applied to the string elements is not specified.

string-map! proc s [start [end]] [Scheme Procedure]
scm_string_map_x (proc, s, start, end) [C Function]
proc is a char->char procedure, it is mapped over s. The order in which the procedure
is applied to the string elements is not specified. The string s is modified in-place,
the return value is not specified.

string-for-each proc s [start [end]] [Scheme Procedure]
scm_string_for_each (proc, s, start, end) [C Function]
proc is mapped over s in left-to-right order. The return value is not specified.

string-for-each-index proc s [start [end]] [Scheme Procedure]
scm_string_for_each_index (proc, s, start, end) [C Function]
Call (proc i) for each index i in s, from left to right.
For example, to change characters to alternately upper and lower case,

\[
\begin{align*}
\text{(define str (string-copy "studly"))} \\
\text{(string-for-each-index} \\
\text{  (lambda (i) } \\
\text{    (string-set! str i } \\
\text{      ((if (even? i) char-upcase char-downcase) } \\
\text{        (string-ref str i)))))} \\
\text{str)} \\
\text{str} \Rightarrow \text{"StUdLy"}
\end{align*}
\]

**string-fold** \( kons \) \( knil \) \( s \) [start [end]]  
\[\text{Scheme Procedure}\]
\[\text{scm_string_fold} (kons, knil, s, start, end)\]  
\[\text{C Function}\]

Fold \( kons \) over the characters of \( s \), with \( knil \) as the terminating element, from left to right. \( kons \) must expect two arguments: The actual character and the last result of \( kons \)' application.

**string-fold-right** \( kons \) \( knil \) \( s \) [start [end]]  
\[\text{Scheme Procedure}\]
\[\text{scm_string_fold_right} (kons, knil, s, start, end)\]  
\[\text{C Function}\]

Fold \( kons \) over the characters of \( s \), with \( knil \) as the terminating element, from right to left. \( kons \) must expect two arguments: The actual character and the last result of \( kons \)' application.

**string-unfold** \( p \) \( f \) \( g \) \( seed \) [base [make_final]]  
\[\text{Scheme Procedure}\]
\[\text{scm_string_unfold} (p, f, g, seed, base, make_final)\]  
\[\text{C Function}\]

- \( g \) is used to generate a series of \( seed \) values from the initial \( seed \): \( seed \), \( (g \cdot seed) \), \( (g \cdot 2 \cdot seed) \), \( (g \cdot 3 \cdot seed) \), ...  
- \( p \) tells us when to stop – when it returns true when applied to one of these \( seed \) values.  
- \( f \) maps each \( seed \) value to the corresponding character in the result string. These chars are assembled into the string in a left-to-right order.  
- \( base \) is the optional initial/leftmost portion of the constructed string; it default to the empty string.  
- \( make \_final \) is applied to the terminal \( seed \) value (on which \( p \) returns true) to produce the final/rightmost portion of the constructed string. The default is nothing extra.

**string-unfold-right** \( p \) \( f \) \( g \) \( seed \) [base [make_final]]  
\[\text{Scheme Procedure}\]
\[\text{scm_string_unfold_right} (p, f, g, seed, base, make_final)\]  
\[\text{C Function}\]

- \( g \) is used to generate a series of \( seed \) values from the initial \( seed \): \( seed \), \( (g \cdot seed) \), \( (g \cdot 2 \cdot seed) \), \( (g \cdot 3 \cdot seed) \), ...  
- \( p \) tells us when to stop – when it returns true when applied to one of these \( seed \) values.  
- \( f \) maps each \( seed \) value to the corresponding character in the result string. These chars are assembled into the string in a right-to-left order.  
- \( base \) is the optional initial/rightmost portion of the constructed string; it default to the empty string.
158 Guile Reference Manual

- `make_final` is applied to the terminal seed value (on which `p` returns true) to produce the final/leftmost portion of the constructed string. It defaults to `(lambda (x))`.

### 6.6.5.12 Miscellaneous String Operations

`xsubstring s from [to [start [end]]]`  
Scheme Procedure

This is the extended substring procedure that implements replicated copying of a substring of some string.

- `s` is a string, `start` and `end` are optional arguments that demarcate a substring of `s`, defaulting to 0 and the length of `s`. Replicate this substring up and down index space, in both the positive and negative directions. `xsubstring` returns the substring of this string beginning at index `from`, and ending at `to`, which defaults to `from + (end - start)`.

`string-xcopy! target tstart s from [sto [start [end]]]`  
Scheme Procedure

Exactly the same as `xsubstring`, but the extracted text is written into the string `target` starting at index `tstart`. The operation is not defined if `(eq? target s)` or these arguments share storage – you cannot copy a string on top of itself.

`string-replace s1 s2 [start1 [end1 [start2 [end2]]]]`  
Scheme Procedure

Return the string `s1`, but with the characters `start1 . . . end1` replaced by the characters `start2 . . . end2` from `s2`.

`string-tokenize s [token_set [start [end]]]`  
Scheme Procedure

Split 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`, which defaults to `char-set:graphic`. If `start` or `end` indices are provided, they restrict `string-tokenize` to operating on the indicated substring of `s`.

`string-filter char_pred s [start [end]]`  
Scheme Procedure

Filter the string `s`, retaining only those characters which satisfy `char_pred`.

- If `char_pred` is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

`string-delete char_pred s [start [end]]`  
Scheme Procedure

Delete characters satisfying `char_pred` from `s`.

- If `char_pred` is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.
6.6.5.13 Representing Strings as Bytes

Out in the cold world outside of Guile, not all strings are treated in the same way. Out there there are only bytes, and there are many ways of representing a strings (sequences of characters) as binary data (sequences of bytes).

As a user, usually you don’t have to think about this very much. When you type on your keyboard, your system encodes your keystrokes as bytes according to the locale that you have configured on your computer. Guile uses the locale to decode those bytes back into characters – hopefully the same characters that you typed in.

All is not so clear when dealing with a system with multiple users, such as a web server. Your web server might get a request from one user for data encoded in the ISO-8859-1 character set, and then another request from a different user for UTF-8 data.

Guile provides an iconv module for converting between strings and sequences of bytes. See Section 6.6.12 [Bytevectors], page 193, for more on how Guile represents raw byte sequences. This module gets its name from the common UNIX command of the same name.

Note that often it is sufficient to just read and write strings from ports instead of using these functions. To do this, specify the port encoding using set-port-encoding!. See Section 6.14.1 [Ports], page 328, for more on ports and character encodings.

Unlike the rest of the procedures in this section, you have to load the iconv module before having access to these procedures:

```
(use-modules (ice-9 iconv))
```

`string->bytevector string encoding [conversion-strategy]` [Scheme Procedure]

Encode `string` as a sequence of bytes.

The string will be encoded in the character set specified by the `encoding` string. If the string has characters that cannot be represented in the encoding, by default this procedure raises an `encoding-error`. Pass a `conversion-strategy` argument to specify other behaviors.

The return value is a bytevector. See Section 6.6.12 [Bytevectors], page 193, for more on bytevectors. See Section 6.14.1 [Ports], page 328, for more on character encodings and conversion strategies.

`bytevector->string bytevector encoding [conversion-strategy]` [Scheme Procedure]

Decode `bytevector` into a string.

The bytes will be decoded from the character set by the `encoding` string. If the bytes do not form a valid encoding, by default this procedure raises an `decoding-error`. As with `string->bytevector`, pass the optional `conversion-strategy` argument to modify this behavior. See Section 6.14.1 [Ports], page 328, for more on character encodings and conversion strategies.

`call-with-output-encoded-string encoding proc [conversion-strategy]` [Scheme Procedure]

Like `call-with-output-string`, but instead of returning a string, returns a encoding of the string according to `encoding`, as a bytevector. This procedure can be more efficient than collecting a string and then converting it via `string->bytevector`. 
6.6.5.14 Conversion to/from C

When creating a Scheme string from a C string or when converting a Scheme string to a C string, the concept of character encoding becomes important.

In C, a string is just a sequence of bytes, and the character encoding describes the relation between these bytes and the actual characters that make up the string. For Scheme strings, character encoding is not an issue (most of the time), since in Scheme you usually treat strings as character sequences, not byte sequences.

Converting to C and converting from C each have their own challenges.

When converting from C to Scheme, it is important that the sequence of bytes in the C string be valid with respect to its encoding. ASCII strings, for example, can’t have any bytes greater than 127. An ASCII byte greater than 127 is considered *ill-formed* and cannot be converted into a Scheme character.

Problems can occur in the reverse operation as well. Not all character encodings can hold all possible Scheme characters. Some encodings, like ASCII for example, can only describe a small subset of all possible characters. So, when converting to C, one must first decide what to do with Scheme characters that can’t be represented in the C string.

Converting a Scheme string to a C string will often allocate fresh memory to hold the result. You must take care that this memory is properly freed eventually. In many cases, this can be achieved by using *scm_dynwind_free* inside an appropriate dynwind context, See Section 6.13.10 [Dynamic Wind], page 317.

```c
SCM scm_from_locale_string (const char *str)  [C Function]
SCM scm_from_locale_stringn (const char *str, size_t len)  [C Function]
```

Creates a new Scheme string that has the same contents as *str* when interpreted in the character encoding of the current locale.

For *scm_from_locale_string*, *str* must be null-terminated.

For *scm_from_locale_stringn*, *len* specifies the length of *str* in bytes, and *str* does not need to be null-terminated. If *len* is *(size_t)-1*, then *str* does need to be null-terminated and the real length will be found with *strlen*.

If the C string is ill-formed, an error will be raised.

Note that these functions should *not* be used to convert C string constants, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so to convert C string constants we recommend *scm_from_utf8_string*.

```c
SCM scm_take_locale_string (char *str)  [C Function]
SCM scm_take_locale_stringn (char *str, size_t len)  [C Function]
```

Like *scm_from_locale_string* and *scm_from_locale_stringn*, respectively, but also frees *str* with *free* eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.
char * scm_to_locale_string (SCM str) [C Function]
char * scm_to_locale_stringn (SCM str, size_t *lenp) [C Function]

Returns a C string with the same contents as str in the character encoding of the current locale. The C string must be freed with free eventually, maybe by using scm_dynwind_free, See Section 6.13.10 [Dynamic Wind], page 317.

For scm_to_locale_string, the returned string is null-terminated and an error is signalled when str contains #\null characters.

For scm_to_locale_stringn and lenp not NULL, str might contain #\null characters and the length of the returned string in bytes is stored in *lenp. The returned string will not be null-terminated in this case. If lenp is NULL, scm_to_locale_stringn behaves like scm_to_locale_string.

If a character in str cannot be represented in the character encoding of the current locale, the default port conversion strategy is used. See Section 6.14.1 [Ports], page 328, for more on conversion strategies.

If the conversion strategy is error, an error will be raised. If it is substitute, a replacement character, such as a question mark, will be inserted in its place. If it is escape, a hex escape will be inserted in its place.

size_t scm_to_locale_stringbuf (SCM str, char *buf, size_t max_len) [C Function]

Puts str as a C string in the current locale encoding into the memory pointed to by buf. The buffer at buf has room for max_len bytes and scm_to_local_stringbuf will never store more than that. No terminating ‘\0’ will be stored.

The return value of scm_to_locale_stringbuf is the number of bytes that are needed for all of str, regardless of whether buf was large enough to hold them. Thus, when the return value is larger than max_len, only max_len bytes have been stored and you probably need to try again with a larger buffer.

For most situations, string conversion should occur using the current locale, such as with the functions above. But there may be cases where one wants to convert strings from a character encoding other than the locale’s character encoding. For these cases, the lower-level functions scm_to_stringn and scm_from_stringn are provided. These functions should seldom be necessary if one is properly using locales.

scm_t_string_failed_conversion_handler [C Type]

This is an enumerated type that can take one of three values: SCM_FAILED_CONVERSION_ERROR, SCM_FAILED_CONVERSION_QUESTION_MARK, and SCM_FAILED_CONVERSION_ESCAPE_SEQUENCE. They are used to indicate a strategy for handling characters that cannot be converted to or from a given character encoding. SCM_FAILED_CONVERSION_ERROR indicates that a conversion should throw an error if some characters cannot be converted. SCM_FAILED_CONVERSION_QUESTION_MARK indicates that a conversion should replace unconvertable characters with the question mark character. And, SCM_FAILED_CONVERSION_ESCAPE_SEQUENCE requests that a conversion should replace an unconvertable character with an escape sequence.
While all three strategies apply when converting Scheme strings to C, only \texttt{SCM}{}\_\texttt{FAILED}{}\_\texttt{CONVERSION}{}\_\texttt{ERROR} and \texttt{SCM}{}\_\texttt{FAILED}{}\_\texttt{CONVERSION}{}\_\texttt{QUESTION}{}\_\texttt{MARK} can be used when converting C strings to Scheme.

\begin{tabular}{|l|}
\hline
\texttt{char *scm\_to\_stringn (SCM str, size\_t *lenp, const char *encoding, scm\_t\_string\_failed\_conversion\_handler handler)}
\texttt{[C Function]}
\hline
This function returns a newly allocated C string from the Guile string \texttt{str}. The length of the returned string in bytes will be returned in \texttt{lenp}. The character encoding of the C string is passed as the ASCII, null-terminated C string \texttt{encoding}. The \texttt{handler} parameter gives a strategy for dealing with characters that cannot be converted into \texttt{encoding}. If \texttt{lenp} is \texttt{NULL}, this function will return a null-terminated C string. It will throw an error if the string contains a null character.

The Scheme interface to this function is \texttt{string->bytevector}, from the \texttt{ice-9} \texttt{iconv} module. See Section 6.6.5.13 [Representing Strings as Bytes], page 159.

\begin{tabular}{|l|}
\hline
\texttt{SCM scm\_from\_stringn (const char *str, size\_t len, const char *encoding, scm\_t\_string\_failed\_conversion\_handler handler)}
\texttt{[C Function]}
\hline
This function returns a scheme string from the C string \texttt{str}. The length in bytes of the C string is input as \texttt{len}. The encoding of the C string is passed as the ASCII, null-terminated C string \texttt{encoding}. The \texttt{handler} parameters suggest a strategy for dealing with unconvertable characters.

The Scheme interface to this function is \texttt{bytevector->string}. See Section 6.6.5.13 [Representing Strings as Bytes], page 159.

The following conversion functions are provided as a convenience for the most commonly used encodings.

\begin{tabular}{|l|}
\hline
\texttt{SCM scm\_from\_latin1\_string (const char *str)}
\texttt{[C Function]}
\hline
\texttt{SCM scm\_from\_utf8\_string (const char *str)}
\texttt{[C Function]}
\hline
\texttt{SCM scm\_from\_utf32\_string (const wchar *str)}
\texttt{[C Function]}
\hline
Return a scheme string from the null-terminated C string \texttt{str}, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded. These functions should be used to convert hard-coded C string constants into Scheme strings.

\begin{tabular}{|l|}
\hline
\texttt{SCM scm\_from\_latin1\_stringn (const char *str, size\_t len)}
\texttt{[C Function]}
\hline
\texttt{SCM scm\_from\_utf8\_stringn (const char *str, size\_t len)}
\texttt{[C Function]}
\hline
\texttt{SCM scm\_from\_utf32\_stringn (const wchar *str, size\_t len)}
\texttt{[C Function]}
\hline
Return a scheme string from C string \texttt{str}, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded, of length \texttt{len}. \texttt{len} is the number of bytes pointed to by \texttt{str} for \texttt{scm\_from\_latin1\_stringn} and \texttt{scm\_from\_utf8\_stringn}; it is the number of elements (code points) in \texttt{str} in the case of \texttt{scm\_from\_utf32\_stringn}.

\begin{tabular}{|l|}
\hline
\texttt{char *scm\_to\_latin1\_stringn (SCM str, size\_t *lenp)}
\texttt{[C function]}
\hline
\texttt{char *scm\_to\_utf8\_stringn (SCM str, size\_t *lenp)}
\texttt{[C function]}
\hline
\texttt{wchar *scm\_to\_utf32\_stringn (SCM str, size\_t *lenp)}
\texttt{[C function]}
\hline
Return a newly allocated, ISO-8859-1-, UTF-8-, or UTF-32-encoded C string from Scheme string \texttt{str}. An error is thrown when \texttt{str} cannot be converted to the specified
encoding. If `lenp` is `NULL`, the returned C string will be null terminated, and an error will be thrown if the C string would otherwise contain null characters. If `lenp` is not `NULL`, the string is not null terminated, and the length of the returned string is returned in `lenp`. The length returned is the number of bytes for `scm_to_latin1_stringn` and `scm_to_utf8_stringn`; it is the number of elements (code points) for `scm_to_utf32_stringn`.

It is not often the case, but sometimes when you are dealing with the implementation details of a port, you need to encode and decode strings according to the encoding and conversion strategy of the port. There are some convenience functions for that purpose as well.

```c
SCM scm_from_port_string (const char *str, SCM port)  [C Function]
SCM scm_from_port_stringn (const char *str, size_t len, SCM port)  [C Function]
char* scm_to_port_string (SCM str, SCM port)  [C Function]
char* scm_to_port_stringn (SCM str, size_t *lenp, SCM port)  [C Function]
```

Like `scm_from_stringn` and friends, except they take their encoding and conversion strategy from a given port object.

### 6.6.5.15 String Internals

Guile stores each string in memory as a contiguous array of Unicode code points along with an associated set of attributes. If all of the code points of a string have an integer range between 0 and 255 inclusive, the code point array is stored as one byte per code point: it is stored as an ISO-8859-1 (aka Latin-1) string. If any of the code points of the string has an integer value greater that 255, the code point array is stored as four bytes per code point: it is stored as a UTF-32 string.

Conversion between the one-byte-per-code-point and four-bytes-per-code-point representations happens automatically as necessary.

No API is provided to set the internal representation of strings; however, there are pair of procedures available to query it. These are debugging procedures. Using them in production code is discouraged, since the details of Guile’s internal representation of strings may change from release to release.

```scheme
(string-bytes-per-char str)  [Scheme Procedure]
scm_string_bytes_per_char (str)  [C Function]
```

Return the number of bytes used to encode a Unicode code point in string `str`. The result is one or four.

```scheme
(%string-dump str)  [Scheme Procedure]
scm_sys_string_dump (str)  [C Function]
```

Returns an association list containing debugging information for `str`. The association list has the following entries.

- `string`: The string itself.
- `start`: The start index of the string into its stringbuf
- `length`: The length of the string
shared     If this string is a substring, it returns its parent string. Otherwise, it returns #f

read-only   #t if the string is read-only

stringbuf-chars
A new string containing this string’s stringbuf’s characters

stringbuf-length
The number of characters in this stringbuf

stringbuf-shared
#t if this stringbuf is shared

stringbuf-wide
#t if this stringbuf’s characters are stored in a 32-bit buffer, or #f if they are stored in an 8-bit buffer

6.6.6 Symbols
Symbols in Scheme are widely used in three ways: as items of discrete data, as lookup keys for alists and hash tables, and to denote variable references.

A symbol is similar to a string in that it is defined by a sequence of characters. The sequence of characters is known as the symbol’s name. In the usual case — that is, where the symbol’s name doesn’t include any characters that could be confused with other elements of Scheme syntax — a symbol is written in a Scheme program by writing the sequence of characters that make up the name, without any quotation marks or other special syntax. For example, the symbol whose name is “multiply-by-2” is written, simply:

   multiply-by-2

Notice how this differs from a string with contents “multiply-by-2”, which is written with double quotation marks, like this:

   "multiply-by-2"

Looking beyond how they are written, symbols are different from strings in two important respects.

The first important difference is uniqueness. If the same-looking string is read twice from two different places in a program, the result is two different string objects whose contents just happen to be the same. If, on the other hand, the same-looking symbol is read twice from two different places in a program, the result is the same symbol object both times.

Given two read symbols, you can use eq? to test whether they are the same (that is, have the same name). eq? is the most efficient comparison operator in Scheme, and comparing two symbols like this is as fast as comparing, for example, two numbers. Given two strings, on the other hand, you must use equal? or string=?, which are much slower comparison operators, to determine whether the strings have the same contents.

   (define sym1 (quote hello))
   (define sym2 (quote hello))
   (eq? sym1 sym2) ⇒ #t

   (define str1 "hello")
(define str2 "hello")
(eq? str1 str2) ⇒ #f
(equal? str1 str2) ⇒ #t

The second important difference is that symbols, unlike strings, are not self-evaluating. This is why we need the (quote ...)s in the example above: (quote hello) evaluates to the symbol named "hello" itself, whereas an unquoted hello is read as the symbol named "hello" and evaluated as a variable reference . . . about which more below (see Section 6.6.6.3 [Symbol Variables], page 167).

6.6.6.1 Symbols as Discrete Data

Numbers and symbols are similar to the extent that they both lend themselves to eq? comparison. But symbols are more descriptive than numbers, because a symbol’s name can be used directly to describe the concept for which that symbol stands.

For example, imagine that you need to represent some colours in a computer program. Using numbers, you would have to choose arbitrarily some mapping between numbers and colours, and then take care to use that mapping consistently:

;;; 1=red, 2=green, 3=purple

(if (eq? (colour-of vehicle) 1)
  ...
)

You can make the mapping more explicit and the code more readable by defining constants:

(define red 1)
(define green 2)
(define purple 3)

(if (eq? (colour-of vehicle) red)
  ...
)

But the simplest and clearest approach is not to use numbers at all, but symbols whose names specify the colours that they refer to:

(if (eq? (colour-of vehicle) 'red)
  ...
)

The descriptive advantages of symbols over numbers increase as the set of concepts that you want to describe grows. Suppose that a car object can have other properties as well, such as whether it has or uses:

- automatic or manual transmission
- leaded or unleaded fuel
- power steering (or not).

Then a car’s combined property set could be naturally represented and manipulated as a list of symbols:

(properties-of vehicle1) ⇒
(red manual unleaded power-steering)

(if (memq 'power-steering (properties-of vehicle1))
Unfit people can drive this vehicle.

Remember, the fundamental property of symbols that we are relying on here is that an occurrence of 'red in one part of a program is an indistinguishable symbol from an occurrence of 'red in another part of a program; this means that symbols can usefully be compared using eq?. At the same time, symbols have naturally descriptive names. This combination of efficiency and descriptive power makes them ideal for use as discrete data.

### 6.6.6.2 Symbols as Lookup Keys

Given their efficiency and descriptive power, it is natural to use symbols as the keys in an association list or hash table.

To illustrate this, consider a more structured representation of the car properties example from the preceding subsection. Rather than mixing all the properties up together in a flat list, we could use an association list like this:

```scheme
(define car1-properties '((colour . red) (transmission . manual) (fuel . unleaded) (steering . power-assisted)))
```

Notice how this structure is more explicit and extensible than the flat list. For example it makes clear that manual refers to the transmission rather than, say, the windows or the locking of the car. It also allows further properties to use the same symbols among their possible values without becoming ambiguous:

```scheme
(define car1-properties '((colour . red) (transmission . manual) (fuel . unleaded) (steering . power-assisted) (seat-colour . red) (locking . manual)))
```

With a representation like this, it is easy to use the efficient assq-XXX family of procedures (see Section 6.6.20 [Association Lists], page 229) to extract or change individual pieces of information:

```scheme
(assq-ref car1-properties 'fuel) ⇒ unleaded
(assq-ref car1-properties 'transmission) ⇒ manual

(assq-set! car1-properties 'seat-colour 'black)
⇒
  ((colour . red)
   (transmission . manual)
   (fuel . unleaded)
   (steering . power-assisted)
   (seat-colour . black)
   (locking . manual)))
```
Hash tables also have keys, and exactly the same arguments apply to the use of symbols in hash tables as in association lists. The hash value that Guile uses to decide where to add a symbol-keyed entry to a hash table can be obtained by calling the `symbol-hash` procedure:

```lang-scheme
symbol-hash symbol
```

Return a hash value for `symbol`.

See Section 6.6.22 [Hash Tables], page 237, for information about hash tables in general, and for why you might choose to use a hash table rather than an association list.

### 6.6.6.3 Symbols as Denoting Variables

When an unquoted symbol in a Scheme program is evaluated, it is interpreted as a variable reference, and the result of the evaluation is the appropriate variable's value.

For example, when the expression `(string-length "abcd")` is read and evaluated, the sequence of characters `string-length` is read as the symbol whose name is "string-length". This symbol is associated with a variable whose value is the procedure that implements string length calculation. Therefore evaluation of the `string-length` symbol results in that procedure.

The details of the connection between an unquoted symbol and the variable to which it refers are explained elsewhere. See Section 6.12 [Binding Constructs], page 293, for how associations between symbols and variables are created, and Section 6.20 [Modules], page 406, for how those associations are affected by Guile's module system.

### 6.6.6.4 Operations Related to Symbols

Given any Scheme value, you can determine whether it is a symbol using the `symbol?` primitive:

```lang-scheme
symbol? obj
```

Return `#t` if `obj` is a symbol, otherwise return `#f`.

```c
int scm_is_symbol (SCM val)
```

Equivalent to `scm_is_true (scm_symbol_p (val))`.

Once you know that you have a symbol, you can obtain its name as a string by calling `symbol->string`. Note that Guile differs by default from R5RS on the details of `symbol->string` as regards case-sensitivity:

```lang-scheme
symbol->string s
```

Return the name of symbol `s` as a string. By default, Guile reads symbols case-sensitively, so the string returned will have the same case variation as the sequence of characters that caused `s` to be created.

If Guile is set to read symbols case-insensitively (as specified by R5RS), and `s` comes into being as part of a literal expression (see Section “Literal expressions” in The
Revised 5 Report on Scheme) or by a call to the read or string-ci->symbol procedures, Guile converts any alphabetic characters in the symbol’s name to lower case before creating the symbol object, so the string returned here will be in lower case.

If s was created by string->symbol, the case of characters in the string returned will be the same as that in the string that was passed to string->symbol, regardless of Guile’s case-sensitivity setting at the time s was created.

It is an error to apply mutation procedures like string-set! to strings returned by this procedure.

Most symbols are created by writing them literally in code. However it is also possible to create symbols programmatically using the following procedures:

**symbol char...**  
* [Scheme Procedure]  
Return a newly allocated symbol made from the given character arguments.

```scheme
(symbol #\x #\y #\z) ⇒ xyz
```

**list->symbol lst**  
* [Scheme Procedure]  
Return a newly allocated symbol made from a list of characters.

```scheme
(list->symbol '(#\a #\b #\c)) ⇒ abc
```

**symbol-append arg ...**  
* [Scheme Procedure]  
Return a newly allocated symbol whose characters form the concatenation of the given symbols, arg ....

```scheme
(let ((h 'hello))
  (symbol-append h 'world))
⇒ helloworld
```

**string->symbol string**  
* [Scheme Procedure]  
Return the symbol whose name is string. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves.

**string-ci->symbol str**  
* [Scheme Procedure]  
Return the symbol whose name is str. If Guile is currently reading symbols case-insensitively, str is converted to lowercase before the returned symbol is looked up or created.

The following examples illustrate Guile’s detailed behaviour as regards the case-sensitivity of symbols:

```scheme
(read-enable 'case-insensitive) ; R5RS compliant behaviour

(symbol->string 'flying-fish) ⇒ "flying-fish"
(symbol->string 'Martin) ⇒ "martin"
(symbol->string (string->symbol "Malvina")) ⇒ "Malvina"
```
(eq? 'mISSISSIppi 'mississippi) ⇒ #t
(string->symbol "mISSISSIppi") ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #f
(eq? 'LolliPop
   (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D.
   (symbol->string
      (string->symbol "K. Harper, M.D."))) ⇒ #t

(read-disable 'case-insensitive) ; Guile default behaviour

(symbol->string 'flying-fish) ⇒ "flying-fish"
(symbol->string 'Martin) ⇒ "Martin"
(symbol->string
   (string->symbol "Malvina")) ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi) ⇒ #f
(string->symbol "mISSISSIppi") ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #t
(eq? 'LolliPop
   (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D.
   (symbol->string
      (string->symbol "K. Harper, M.D."))) ⇒ #t

From C, there are lower level functions that construct a Scheme symbol from a C string in the current locale encoding.

When you want to do more from C, you should convert between symbols and strings using scm_symbol_to_string and scm_string_to_symbol and work with the strings.

SCM scm_from_latin1_symbol (const char *name) [C Function]
SCM scm_from_utf8_symbol (const char *name) [C Function]
   Construct and return a Scheme symbol whose name is specified by the null-terminated C string name. These are appropriate when the C string is hard-coded in the source code.

SCM scm_from_locale_symbol (const char *name) [C Function]
SCM scm_from_locale_symboln (const char *name, size_t len) [C Function]
   Construct and return a Scheme symbol whose name is specified by name. For scm_from_locale_symbol, name must be null terminated; for scm_from_locale_symboln the length of name is specified explicitly by len.

Note that these functions should not be used when name is a C string constant, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so in such cases we recommend scm_from_utf8_symbol.
SCM scm_take_locale_symbol (char *str)  
SCM scm_take_locale_symboln (char *str, size_t len)  
Like scm_from_locale_symbol and scm_from_locale_symboln, respectively, but also frees str with free eventually. Thus, you can use this function when you would free str anyway immediately after creating the Scheme string. In certain cases, Guile can then use str directly as its internal representation.

The size of a symbol can also be obtained from C:

size_t scm_c_symbol_length (SCM sym)  
Return the number of characters in sym.

Finally, some applications, especially those that generate new Scheme code dynamically, need to generate symbols for use in the generated code. The gensym primitive meets this need:

gensym [prefix]  
scm_gensym (prefix)  
Create a new symbol with a name constructed from a prefix and a counter value. The string prefix can be specified as an optional argument. Default prefix is ‘g’. The counter is increased by 1 at each call. There is no provision for resetting the counter.

The symbols generated by gensym are likely to be unique, since their names begin with a space and it is only otherwise possible to generate such symbols if a programmer goes out of their way to do so. Uniqueness can be guaranteed by instead using uninterned symbols (see Section 6.6.6.7 [Symbol Uninterned], page 172), though they can’t be usefully written out and read back in.

6.6.6.5 Function Slots and Property Lists

In traditional Lisp dialects, symbols are often understood as having three kinds of value at once:

- a variable value, which is used when the symbol appears in code in a variable reference context
- a function value, which is used when the symbol appears in code in a function name position (i.e. as the first element in an unquoted list)
- a property list value, which is used when the symbol is given as the first argument to Lisp’s put or get functions.

Although Scheme (as one of its simplifications with respect to Lisp) does away with the distinction between variable and function namespaces, Guile currently retains some elements of the traditional structure in case they turn out to be useful when implementing translators for other languages, in particular Emacs Lisp.

Specifically, Guile symbols have two extra slots, one for a symbol’s property list, and one for its “function value.” The following procedures are provided to access these slots.

symbol-fref symbol  
scm_symbol_fref (symbol)  
Return the contents of symbol’s function slot.
symbol-fset! symbol value

 scm_symbol_fset_x (symbol, value)  [C Function]
    Set the contents of symbol's function slot to value.

symbol-pref symbol

 scm_symbol_pref (symbol)  [C Function]
    Return the property list currently associated with symbol.

symbol-pset! symbol value

 scm_symbol_pset_x (symbol, value)  [C Function]
    Set symbol's property list to value.

symbol-property sym prop

 scm_symbol_property (sym, prop)  [C Function]
    From sym's property list, return the value for property prop. The assumption is that
    sym's property list is an association list whose keys are distinguished from each other
    using equal?. prop should be one of the keys in that list. If the property list has no
    entry for prop, symbol-property returns #f.

set-symbol-property! sym prop val

 scm_symbol_property_set (sym, prop, val)  [C Function]
    In sym's property list, set the value for property prop to val, or add a new entry for
    prop, with value val, if none already exists. For the structure of the property list, see
    symbol-property.

symbol-property-remove! sym prop

 scm_symbol_property_remove (sym, prop)  [C Function]
    From sym's property list, remove the entry for property prop, if there is one. For the
    structure of the property list, see symbol-property.

Support for these extra slots may be removed in a future release, and it is probably
better to avoid using them. For a more modern and Schemely approach to properties, see
Section 6.11.2 [Object Properties], page 285.

6.6.6.6 Extended Read Syntax for Symbols

The read syntax for a symbol is a sequence of letters, digits, and extended alphabetic
characters, beginning with a character that cannot begin a number. In addition, the special
cases of +, -, and . . . are read as symbols even though numbers can begin with +, - or .

Extended alphabetic characters may be used within identifiers as if they were letters.

The set of extended alphabetic characters is:

    ! $ % & * + - . / : < > ? @ ^ _ ~

In addition to the standard read syntax defined above (which is taken from R5RS (see
Section “Formal syntax” in The Revised^5 Report on Scheme)), Guile provides an extended
symbol read syntax that allows the inclusion of unusual characters such as space characters,
newlines and parentheses. If (for whatever reason) you need to write a symbol containing
characters not mentioned above, you can do so as follows.

• Begin the symbol with the characters #\{{,}
• write the characters of the symbol and
• finish the symbol with the characters }\#.
Here are a few examples of this form of read syntax. The first symbol needs to use extended syntax because it contains a space character, the second because it contains a line break, and the last because it looks like a number.

#{foo bar}#
#{what
ever}#
#{4242}#

Although Guile provides this extended read syntax for symbols, widespread usage of it is discouraged because it is not portable and not very readable.

Alternatively, if you enable the r7rs-symbols read option (see see Section 6.18.2 [Scheme Read], page 382), you can write arbitrary symbols using the same notation used for strings, except delimited by vertical bars instead of double quotes.

|foo bar|
|\x3BB; is a greek lambda|
|\| is a vertical bar|

Note that there’s also an r7rs-symbols print option (see Section 6.18.3 [Scheme Write], page 383). To enable the use of this notation, evaluate one or both of the following expressions:

(read-enable 'r7rs-symbols)
(print-enable 'r7rs-symbols)

6.6.6.7 Uninterned Symbols

What makes symbols useful is that they are automatically kept unique. There are no two symbols that are distinct objects but have the same name. But of course, there is no rule without exception. In addition to the normal symbols that have been discussed up to now, you can also create special uninterned symbols that behave slightly differently.

To understand what is different about them and why they might be useful, we look at how normal symbols are actually kept unique.

Whenever Guile wants to find the symbol with a specific name, for example during read or when executing string->symbol, it first looks into a table of all existing symbols to find out whether a symbol with the given name already exists. When this is the case, Guile just returns that symbol. When not, a new symbol with the name is created and entered into the table so that it can be found later.

Sometimes you might want to create a symbol that is guaranteed ‘fresh’, i.e. a symbol that did not exist previously. You might also want to somehow guarantee that no one else will ever unintentionally stumble across your symbol in the future. These properties of a symbol are often needed when generating code during macro expansion. When introducing new temporary variables, you want to guarantee that they don’t conflict with variables in other people’s code.

The simplest way to arrange for this is to create a new symbol but not enter it into the global table of all symbols. That way, no one will ever get access to your symbol by chance. Symbols that are not in the table are called uninterned. Of course, symbols that are in the table are called interned.
You create new uninterned symbols with the function `make-symbol`. You can test whether a symbol is interned or not with `symbol-interned?`.

Uninterned symbols break the rule that the name of a symbol uniquely identifies the symbol object. Because of this, they can not be written out and read back in like interned symbols. Currently, Guile has no support for reading uninterned symbols. Note that the function `gensym` does not return uninterned symbols for this reason.

### Scheme Procedure

`make-symbol name`

Return a new uninterned symbol with the name `name`. The returned symbol is guaranteed to be unique and future calls to `string->symbol` will not return it.

### Scheme Procedure

`symbol-interned? symbol`

Return `#t` if `symbol` is interned, otherwise return `#f`.

For example:

```scheme
(define foo-1 (string->symbol "foo"))
(define foo-2 (string->symbol "foo"))
(define foo-3 (make-symbol "foo"))
(define foo-4 (make-symbol "foo"))

(eq? foo-1 foo-2) ⇒ #t
⇒ Two interned symbols with the same name are the same object,

(eq? foo-1 foo-3) ⇒ #f
⇒ but a call to `make-symbol` with the same name returns a distinct object.

(eq? foo-3 foo-4) ⇒ #f
⇒ A call to `make-symbol` always returns a new object, even for the same name.

foo-3 ⇒ #<uninterned-symbol foo 8085290>
⇒ Uninterned symbols print differently from interned symbols,

(symbol? foo-3) ⇒ #t
⇒ but they are still symbols,

(symbol-interned? foo-3) ⇒ #f
⇒ just not interned.
```
6.6.7 Keywords

Keywords are self-evaluating objects with a convenient read syntax that makes them easy to type.

Guile’s keyword support conforms to R5RS, and adds a (switchable) read syntax extension to permit keywords to begin with : as well as #: or to end with :.

6.6.7.1 Why Use Keywords?

Keywords are useful in contexts where a program or procedure wants to be able to accept a large number of optional arguments without making its interface unmanageable.

To illustrate this, consider a hypothetical `make-window` procedure, which creates a new window on the screen for drawing into using some graphical toolkit. There are many parameters that the caller might like to specify, but which could also be sensibly defaulted, for example:

- color depth – Default: the color depth for the screen
- background color – Default: white
- width – Default: 600
- height – Default: 400

If `make-window` did not use keywords, the caller would have to pass in a value for each possible argument, remembering the correct argument order and using a special value to indicate the default value for that argument:

```
(make-window 'default ;; Color depth 'default ;; Background color
   800 ;; Width
   100 ;; Height
   ...) ;; More make-window arguments
```

With keywords, on the other hand, defaulted arguments are omitted, and non-default arguments are clearly tagged by the appropriate keyword. As a result, the invocation becomes much clearer:

```
(make-window #:width 800 #:height 100)
```

On the other hand, for a simpler procedure with few arguments, the use of keywords would be a hindrance rather than a help. The primitive procedure `cons`, for example, would not be improved if it had to be invoked as

```
(cons #:car x #:cdr y)
```

So the decision whether to use keywords or not is purely pragmatic: use them if they will clarify the procedure invocation at point of call.

6.6.7.2 Coding With Keywords

If a procedure wants to support keywords, it should take a rest argument and then use whatever means is convenient to extract keywords and their corresponding arguments from the contents of that rest argument.

The following example illustrates the principle: the code for `make-window` uses a helper procedure called `get-keyword-value` to extract individual keyword arguments from the rest argument.

```
(define (get-keyword-value args keyword default)
  ...)
```
(let ((kv (memq keyword args)))
  (if (and kv (>= (length kv) 2))
      (cadr kv)
      default)))

(define (make-window . args)
  (let ((depth (get-keyword-value args #:depth screen-depth))
        (bg (get-keyword-value args #:bg "white"))
        (width (get-keyword-value args #:width 800))
        (height (get-keyword-value args #:height 100)))
    ...)
  ...
)

But you don’t need to write `get-keyword-value`. The `(ice-9 optargs)` module provides a set of powerful macros that you can use to implement keyword-supporting procedures like this:

(use-modules (ice-9 optargs))

(define (make-window . args)
  (let-keywords args #f ((depth screen-depth)
                          (bg "white")
                          (width 800)
                          (height 100)))
  ...
)

Or, even more economically, like this:

(use-modules (ice-9 optargs))

(define* (make-window #:key (depth screen-depth)
                     (bg "white")
                     (width 800)
                     (height 100)))
  ...
)

For further details on `let-keywords`, `define*` and other facilities provided by the `(ice-9 optargs)` module, see Section 6.9.4 [Optional Arguments], page 252.

To handle keyword arguments from procedures implemented in C, use `scm_c_bind_keyword_arguments` (see Section 6.6.7.4 [Keyword Procedures], page 176).

### 6.6.7.3 Keyword Read Syntax

Guile, by default, only recognizes a keyword syntax that is compatible with R5RS. A token of the form `#:NAME`, where `NAME` has the same syntax as a Scheme symbol (see Section 6.6.6.6 [Symbol Read Syntax], page 171), is the external representation of the keyword named `NAME`. Keyword objects print using this syntax as well, so values containing keyword objects can be read back into Guile. When used in an expression, keywords are self-quoting objects.

If the `keywords` read option is set to `prefix`, Guile also recognizes the alternative read syntax `:NAME`. Otherwise, tokens of the form `:NAME` are read as symbols, as required by R5RS.
If the `keywords` read option is set to `postfix`, Guile recognizes the SRFI-88 read syntax `NAME:` (see Section 7.5.42 [SRFI-88], page 646). Otherwise, tokens of this form are read as symbols.

To enable and disable the alternative non-R5RS keyword syntax, you use the `read-set!` procedure documented Section 6.18.2 [Scheme Read], page 382. Note that the `prefix` and `postfix` syntax are mutually exclusive.

```scheme
(read-set! keywords 'prefix)
#:type ⇒ #:type
#:type ⇒ #:type
(read-set! keywords 'postfix)
:type: ⇒ #:type
#:type ⇒ :type
(read-set! keywords #f)
#:type ⇒ #:type
#:type ⇒ :type
ERROR: In expression :type:
ERROR: Unbound variable: :type
ABORT: (unbound-variable)
```

### 6.6.7.4 Keyword Procedures

- `keyword? obj`  
  Return `#t` if the argument `obj` is a keyword, else `#f`.

- `scm_keyword_p (obj)`  
  [C Function]

- `keyword->symbol keyword`  
  Return the symbol with the same name as `keyword`.

- `scm_keyword_to_symbol (keyword)`  
  [C Function]
symbol->keyword symbol

scm_symbol_to_keyword (symbol)

Return the keyword with the same name as symbol.

int scm_is_keyword (SCM obj)

Equivalent to scm_is_true (scm_keyword_p (obj)).

SCM scm_from_locale_keyword (const char *name)

SCM scm_from_locale_keywordn (const char *name, size_t len)

Equivalent to scm_symbol_to_keyword (scm_from_locale_symbol (name)) and
scm_symbol_to_keyword (scm_from_locale_symboln (name, len)), respectively.

Note that these functions should not be used when name is a C string constant,
because there is no guarantee that the current locale will match that of the execution
character set, used for string and character constants. Most modern C compilers use
UTF-8 by default, so in such cases we recommend scm_from_utf8_keyword.

SCM scm_from_latin1_keyword (const char *name)

SCM scm_from_utf8_keyword (const char *name)

Equivalent to scm_symbol_to_keyword (scm_from_latin1_symbol (name)) and
scm_symbol_to_keyword (scm_from_utf8_symbol (name)), respectively.

void scm_c_bind_keyword_arguments (const char *subr, SCM rest, scm_t_keyword_arguments_flags flags, SCM keyword1, SCM *argp1,
... , SCM keywordN, SCM *argpN, SCM_UNDEFINED)

Extract the specified keyword arguments from rest, which is not modified. If the
keyword argument keyword1 is present in rest with an associated value, that value is
stored in the variable pointed to by argp1, otherwise the variable is left unchanged.
Similarly for the other keywords and argument pointers up to keywordN and argpN.
The argument list to scm_c_bind_keyword_arguments must be terminated by SCM_UNDEFINED.

Note that since the variables pointed to by argp1 through argpN are left unchanged
if the associated keyword argument is not present, they should be initialized to their
default values before calling scm_c_bind_keyword_arguments. Alternatively, you
can initialize them to SCM_UNDEFINED before the call, and then use SCM_UNBNDP after
the call to see which ones were provided.

If an unrecognized keyword argument is present in rest and flags does not contain
SCM_ALLOW_OTHER_KEYS, or if non-keyword arguments are present and flags does not contain
SCM_ALLOW_NON_KEYWORD_ARGUMENTS, an exception is raised. subr should be
the name of the procedure receiving the keyword arguments, for purposes of error
reporting.

For example:

    SCM k_delimiter;
    SCM k_grammar;
    SCM sym_infix;

    SCM my_string_join (SCM strings, SCM rest)
    {
SCM delimiter = SCM_UNDEFINED;
SCM grammar = sym_infix;

scm_c_bind_keyword_arguments ("my-string-join", rest, 0,
k_delimiter, &delimiter,
k_grammar, &grammar,
SCM_UNDEFINED);

if (SCM_UNBNDP (delimiter))
    delimiter = scm_from_utf8_string (" ");

return scm_string_join (strings, delimiter, grammar);
}

void my_init ()
{
    k_delimiter = scm_from_utf8_keyword ("delimiter");
    k_grammar = scm_from_utf8_keyword ("grammar");
    sym_infix = scm_from_utf8_symbol ("infix");
    scm_c_define_gsubr ("my-string-join", 1, 0, 1, my_string_join);
}

### 6.6.8 Pairs

Pairs are used to combine two Scheme objects into one compound object. Hence the name: A pair stores a pair of objects.

The data type pair is extremely important in Scheme, just like in any other Lisp dialect. The reason is that pairs are not only used to make two values available as one object, but that pairs are used for constructing lists of values. Because lists are so important in Scheme, they are described in a section of their own (see Section 6.6.9 [Lists], page 181).

Pairs can literally get entered in source code or at the REPL, in the so-called dotted list syntax. This syntax consists of an opening parentheses, the first element of the pair, a dot, the second element and a closing parentheses. The following example shows how a pair consisting of the two numbers 1 and 2, and a pair containing the symbols foo and bar can be entered. It is very important to write the whitespace before and after the dot, because otherwise the Scheme parser would not be able to figure out where to split the tokens.

```
(1 . 2)
(foo . bar)
```

But beware, if you want to try out these examples, you have to quote the expressions. More information about quotation is available in the section Section 6.18.1.1 [Expression Syntax], page 379. The correct way to try these examples is as follows.

```
'(1 . 2)
⇒
(1 . 2)
'(foo . bar)
⇒
(foo . bar)
```
A new pair is made by calling the procedure cons with two arguments. Then the argument values are stored into a newly allocated pair, and the pair is returned. The name cons stands for "construct". Use the procedure pair? to test whether a given Scheme object is a pair or not.

```
cons x y                         [Scheme Procedure]
scm_cons (x, y)                  [C Function]
    Return a newly allocated pair whose car is x and whose cdr is y. The pair is guaranteed to be different (in the sense of eq?) from every previously existing object.

pair? x                           [Scheme Procedure]
scm_pair_p (x)                    [C Function]
    Return #t if x is a pair; otherwise return #f.

int scm_is_pair (SCM x)           [C Function]
    Return 1 when x is a pair; otherwise return 0.
```

The two parts of a pair are traditionally called car and cdr. They can be retrieved with procedures of the same name (car and cdr), and can be modified with the procedures set-car! and set-cdr!.

Since a very common operation in Scheme programs is to access the car of a car of a pair, or the car of the cdr of a pair, etc., the procedures called caar, cadr and so on are also predefined. However, using these procedures is often detrimental to readability, and error-prone. Thus, accessing the contents of a list is usually better achieved using pattern matching techniques (see Section 7.7 [Pattern Matching], page 691).

```
car pair                        [Scheme Procedure]
cdr pair                        [Scheme Procedure]
scm_car (pair)                  [C Function]
scm_cdr (pair)                  [C Function]
    Return the car or the cdr of pair, respectively.

SCM SCM_CAR (SCM pair)           [C Macro]
SCM SCM_CDR (SCM pair)           [C Macro]
    These two macros are the fastest way to access the car or cdr of a pair; they can be thought of as compiling into a single memory reference.
    These macros do no checking at all. The argument pair must be a valid pair.
```
These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by
caddr, cadr and cadddr pick out the second, third or fourth elements of a list, respectively. SRFI-1 provides the same under the names second, third and fourth (see Section 7.5.3.3 [SRFI-1 Selectors], page 580).

**Scheme Procedure**

```scheme
set-car! pair value
```

Stores value in the car field of pair. The value returned by set-car! is unspecified.

**Scheme Procedure**

```scheme
set-cdr! pair value
```

Stores value in the cdr field of pair. The value returned by set-cdr! is unspecified.

### 6.6.9 Lists

A very important data type in Scheme—as well as in all other Lisp dialects—is the data type list.\(^1\)

This is the short definition of what a list is:

- Either the empty list ()
- or a pair which has a list in its cdr.

#### 6.6.9.1 List Read Syntax

The syntax for lists is an opening parentheses, then all the elements of the list (separated by whitespace) and finally a closing parentheses.\(^2\).

```
(1 2 3) ; a list of the numbers 1, 2 and 3
("foo" bar 3.1415) ; a string, a symbol and a real number
() ; the empty list
```

The last example needs a bit more explanation. A list with no elements, called the empty list, is special in some ways. It is used for terminating lists by storing it into the cdr of the last pair that makes up a list. An example will clear that up:

```
(car '(1))
⇒
1
(cdr '(1))
⇒
()
```

This example also shows that lists have to be quoted when written (see Section 6.18.1.1 [Expression Syntax], page 379), because they would otherwise be mistakenly taken as procedure applications (see Section 3.2.2 [Simple Invocation], page 18).

### 6.6.9.2 List Predicates

Often it is useful to test whether a given Scheme object is a list or not. List-processing procedures could use this information to test whether their input is valid, or they could do different things depending on the datatype of their arguments.

---

\(^1\) Strictly speaking, Scheme does not have a real datatype list. Lists are made up of chained pairs, and only exist by definition—a list is a chain of pairs which looks like a list.

\(^2\) Note that there is no separation character between the list elements, like a comma or a semicolon.
list? x

scm_list_p (x)

Return #t if x is a proper list, else #f.

The predicate null? is often used in list-processing code to tell whether a given list has run out of elements. That is, a loop somehow deals with the elements of a list until the list satisfies null?. Then, the algorithm terminates.

null? x

scm_null_p (x)

Return #t if x is the empty list, else #f.

int scm_is_null (SCM x)

Return 1 when x is the empty list; otherwise return 0.

6.6.9.3 List Constructors

This section describes the procedures for constructing new lists. list simply returns a list where the elements are the arguments, cons* is similar, but the last argument is stored in the cdr of the last pair of the list.

list elem ...

scm_list_1 (elem1)

scm_list_2 (elem1, elem2)

scm_list_3 (elem1, elem2, elem3)

scm_list_4 (elem1, elem2, elem3, elem4)

scm_list_5 (elem1, elem2, elem3, elem4, elem5)

scm_list_n (elem1, ..., elemN, SCM_UNDEFINED)

Return a new list containing elements elem ....

scm_list_n takes a variable number of arguments, terminated by the special SCM_UNDEFINED. That final SCM_UNDEFINED is not included in the list. None of elem ... can themselves be SCM_UNDEFINED, or scm_list_n will terminate at that point.

cons* arg1 arg2 ...

Like list, but the last arg provides the tail of the constructed list, returning (cons arg1 (cons arg2 (cons ... argn))). Requires at least one argument. If given one argument, that argument is returned as result. This function is called list* in some other Schemes and in Common LISP.

list-copy lst

scm_list_copy (lst)

Return a (newly-created) copy of lst.

make-list n [init]

Create a list containing n elements, where each element is initialized to init. init defaults to the empty list () if not given.

Note that list-copy only makes a copy of the pairs which make up the spine of the lists. The list elements are not copied, which means that modifying the elements of the new list also modifies the elements of the old list. On the other hand, applying procedures
like set-cdr! or delv! to the new list will not alter the old list. If you also need to copy the list elements (making a deep copy), use the procedure copy-tree (see Section 6.11.4 [Copying], page 288).

### 6.6.9.4 List Selection

These procedures are used to get some information about a list, or to retrieve one or more elements of a list.

- **length** `lst`  
  [Scheme Procedure]  
  Return the number of elements in list `lst`.

- **last-pair** `lst`  
  [Scheme Procedure]  
  Return the last pair in `lst`, signalling an error if `lst` is circular.

- **list-ref** `list k`  
  [Scheme Procedure]  
  Return the `k`th element from `list`.

- **list-tail** `lst k`  
  [Scheme Procedure]  
  Return the "tail" of `lst` beginning with its `k`th element. The first element of the list is considered to be element 0.

- **list-head** `lst k`  
  [Scheme Procedure]  
  Copy the first `k` elements from `lst` into a new list, and return it.

### 6.6.9.5 Append and Reverse

append and append! are used to concatenate two or more lists in order to form a new list. reverse and reverse! return lists with the same elements as their arguments, but in reverse order. The procedure variants with an ! directly modify the pairs which form the list, whereas the other procedures create new pairs. This is why you should be careful when using the side-effecting variants.

- **append** `lst . . . obj`  
  [Scheme Procedure]  

- **append!** `lst . . . obj`  
  [Scheme Procedure]  

- **scm_append** `(lst lst)`  
  [C Function]  

- **scm_append_x** `(lst lst)`  
  [C Function]  

  Return a list comprising all the elements of lists `lst . . . obj`. If called with no arguments, return the empty list.

  ```scheme
  (append ''(x) ''(y))  =>  (x y)
  ```
The last argument \textit{obj} may actually be any object; an improper list results if the last argument is not a proper list.

\texttt{(append '(a) '(b c d))} \Rightarrow \texttt{(a b c d)}
\texttt{(append '(a (b)) '((c)))} \Rightarrow \texttt{(a (b) (c))}

\texttt{append} doesn’t modify the given lists, but the return may share structure with the final \textit{obj}. \texttt{append!} is permitted, but not required, to modify the given lists to form its return.

For \texttt{scm_append} and \texttt{scm_append_x}, \textit{lstlst} is a list of the list operands \textit{lst} \ldots \textit{obj}. That \textit{lstlst} itself is not modified or used in the return.

\begin{verbatim}
reverse lst
reverse! lst [newtail]
scm_reverse (lst)
scm_reverse_x (lst, newtail)
\end{verbatim}

Return a list comprising the elements of \textit{lst}, in reverse order.

\texttt{reverse} constructs a new list. \texttt{reverse!} is permitted, but not required, to modify \textit{lst} in constructing its return.

For \texttt{reverse!}, the optional \textit{newtail} is appended to the result. \textit{newtail} isn’t reversed, it simply becomes the list tail. For \texttt{scm_reverse_x}, the \textit{newtail} parameter is mandatory, but can be \texttt{SCM_EOL} if no further tail is required.

### 6.6.9.6 List Modification

The following procedures modify an existing list, either by changing elements of the list, or by changing the list structure itself.

\begin{verbatim}
list-set! list k val
scm_list_set_x (list, k, val)
list-cdr-set! list k val
scm_list_cdr_set_x (list, k, val)
delq item lst
scm_delq (item, lst)
delv item lst
scm_delv (item, lst)
delete item lst
scm_delete (item, lst)
\end{verbatim}

Return a newly-created copy of \textit{lst} with elements \texttt{eq?} to \textit{item} removed. This procedure mirrors \texttt{memq}: \texttt{delq} compares elements of \textit{lst} against \textit{item} with \texttt{eq?}.

Return a newly-created copy of \textit{lst} with elements \texttt{eqv?} to \textit{item} removed. This procedure mirrors \texttt{memv}: \texttt{delv} compares elements of \textit{lst} against \textit{item} with \texttt{eqv?}.

Return a newly-created copy of \textit{lst} with elements \texttt{equal?} to \textit{item} removed. This procedure mirrors \texttt{member}: \texttt{delete} compares elements of \textit{lst} against \textit{item} with \texttt{equal?}. 

\texttt{SCM_EOL}
See also SRFI-1 which has an extended `delete` (Section 7.5.3.8 [SRFI-1 Deleting], page 587), and also an `lset-difference` which can delete multiple items in one call (Section 7.5.3.10 [SRFI-1 Set Operations], page 589).

```
[Scheme Procedure] delq! item lst
[Scheme Procedure] delv! item lst
[Scheme Procedure] delete! item lst
[C Function] scm_delq_x (item, lst)
[C Function] scm_delv_x (item, lst)
[C Function] scm_delete_x (item, lst)
```

These procedures are destructive versions of `delq`, `delv` and `delete`: they modify the pointers in the existing `lst` rather than creating a new list. Caveat evaluator: Like other destructive list functions, these functions cannot modify the binding of `lst`, and so cannot be used to delete the first element of `lst` destructively.

```
[Scheme Procedure] delq1! item lst
[C Function] scm_delq1_x (item, lst)
  Like `delq!`, but only deletes the first occurrence of `item` from `lst`. Tests for equality using `eq?`. See also `delv1!` and `delete1!`.

[Scheme Procedure] delv1! item lst
[C Function] scm_delv1_x (item, lst)
  Like `delv!`, but only deletes the first occurrence of `item` from `lst`. Tests for equality using `eqv?`. See also `delq1!` and `delete1!`.

[Scheme Procedure] delete1! item lst
[C Function] scm_delete1_x (item, lst)
  Like `delete!`, but only deletes the first occurrence of `item` from `lst`. Tests for equality using `equal?`. See also `delq1!` and `delv1!`.

[Scheme Procedure] filter pred lst
[Scheme Procedure] filter! pred lst
Return a list containing all elements from `lst` which satisfy the predicate `pred`. The elements in the result list have the same order as in `lst`. The order in which `pred` is applied to the list elements is not specified.

`filter` does not change `lst`, but the result may share a tail with it. `filter!` may modify `lst` to construct its return.

### 6.6.9.7 List Searching

The following procedures search lists for particular elements. They use different comparison predicates for comparing list elements with the object to be searched. When they fail, they return `#f`, otherwise they return the sublist whose car is equal to the search object, where equality depends on the equality predicate used.

```
[Scheme Procedure] memq x lst
[C Function] scm_memq (x, lst)
  Return the first sublist of `lst` whose car is `eq?` to `x` where the sublists of `lst` are the non-empty lists returned by `(list-tail lst k)` for `k` less than the length of `lst`. If `x` does not occur in `lst`, then `#f` (not the empty list) is returned.
```
memv x lst  

scm_memv (x, lst)  

Return the first sublist of lst whose car is `eqv?` to x where the sublists of lst are the non-empty lists returned by `(list-tail lst k)` for k less than the length of lst. If x does not occur in lst, then `#f` (not the empty list) is returned.

member x lst  

scm_member (x, lst)  

Return the first sublist of lst whose car is `equal?` to x where the sublists of lst are the non-empty lists returned by `(list-tail lst k)` for k less than the length of lst. If x does not occur in lst, then `#f` (not the empty list) is returned.

See also SRFI-1 which has an extended `member` function (Section 7.5.3.7 [SRFI-1 Searching], page 586).

6.6.9.8 List Mapping

List processing is very convenient in Scheme because the process of iterating over the elements of a list can be highly abstracted. The procedures in this section are the most basic iterating procedures for lists. They take a procedure and one or more lists as arguments, and apply the procedure to each element of the list. They differ in their return value.

map proc arg1 arg2 ...  

map-in-order proc arg1 arg2 ...  

Apply `proc` to each element of the list arg1 (if only two arguments are given), or to the corresponding elements of the argument lists (if more than two arguments are given). The result(s) of the procedure applications are saved and returned in a list. For `map`, the order of procedure applications is not specified, `map-in-order` applies the procedure from left to right to the list elements.

for-each proc arg1 arg2 ...  

Like `map`, but the procedure is always applied from left to right, and the result(s) of the procedure applications are thrown away. The return value is not specified.

See also SRFI-1 which extends these functions to take lists of unequal lengths (Section 7.5.3.5 [SRFI-1 Fold and Map], page 582).

6.6.10 Vectors

Vectors are sequences of Scheme objects. Unlike lists, the length of a vector, once the vector is created, cannot be changed. The advantage of vectors over lists is that the time required to access one element of a vector given its `position` (synonymous with `index`), a zero-origin number, is constant, whereas lists have an access time linear to the position of the accessed element in the list.

Vectors can contain any kind of Scheme object; it is even possible to have different types of objects in the same vector. For vectors containing vectors, you may wish to use arrays, instead. Note, too, that vectors are the special case of one dimensional non-uniform arrays and that most array procedures operate happily on vectors (see Section 6.6.13 [Arrays], page 200).

Also see Section 7.5.30 [SRFI-43], page 634, for a comprehensive vector library.
6.6.10.1 Read Syntax for Vectors

Vectors can literally be entered in source code, just like strings, characters or some of the other data types. The read syntax for vectors is as follows: A sharp sign (#), followed by an opening parentheses, all elements of the vector in their respective read syntax, and finally a closing parentheses. Like strings, vectors do not have to be quoted.

The following are examples of the read syntax for vectors; where the first vector only contains numbers and the second three different object types: a string, a symbol and a number in hexadecimal notation.

#(1 2 3)
#("Hello" foo #xdeadbeef)

6.6.10.2 Dynamic Vector Creation and Validation

Instead of creating a vector implicitly by using the read syntax just described, you can create a vector dynamically by calling one of the vector and list->vector primitives with the list of Scheme values that you want to place into a vector. The size of the vector thus created is determined implicitly by the number of arguments given.

vector arg ... [Scheme Procedure]
list->vector l [Scheme Procedure]
scm_vector (l) [C Function]
   Return a newly allocated vector composed of the given arguments. Analogous to list.
   (vector 'a 'b 'c) ⇒ #(a b c)

The inverse operation is vector->list:

vector->list v [Scheme Procedure]
scm_vector_to_list (v) [C Function]
   Return a newly allocated list composed of the elements of v.
   (vector->list #(dah dah didah)) ⇒ (dah dah didah)
   (list->vector '(dididit dah)) ⇒ #(dididit dah)

To allocate a vector with an explicitly specified size, use make-vector. With this primitive you can also specify an initial value for the vector elements (the same value for all elements, that is):

make-vector len [fill] [Scheme Procedure]
scm_make_vector (len, fill) [C Function]
   Return a newly allocated vector of len elements. If a second argument is given, then each position is initialized to fill. Otherwise the initial contents of each position is unspecified.

SCM scm_c_make_vector (size_t k, SCM fill) [C Function]
   Like scm_make_vector, but the length is given as a size_t.

To check whether an arbitrary Scheme value is a vector, use the vector? primitive:
vector? obj

scm_vector_p (obj)  
[Scheme Procedure]  
Return #t if obj is a vector, otherwise return #f.

int scm_is_vector (SCM obj)  
[C Function]  
Return non-zero when obj is a vector, otherwise return zero.

### 6.6.10.3 Accessing and Modifying Vector Contents

vector-length and vector-ref return information about a given vector, respectively its size and the elements that are contained in the vector.

**vector-length vector**  
[Scheme Procedure]  
Return the number of elements in vector as an exact integer.

**size_t scm_c_vector_length (SCM vec)**  
[C Function]  
Return the number of elements in vec as a size_t.

**vector-ref vec k**  
[Scheme Procedure]  
Return the contents of position k of vec. k must be a valid index of vec.

(vector-ref #(1 1 2 3 5 8 13 21) 5) ⇒ 8  
(vector-ref #(1 1 2 3 5 8 13 21)  
(let ((i (round (* 2 (acos -1)))))  
(if (inexact? i)  
(inexact->exact i)  
(i))) ⇒ 13

**SCM scm_c_vector_ref (SCM vec, size_t k)**  
[C Function]  
Return the contents of position k (a size_t) of vec.

A vector created by one of the dynamic vector constructor procedures (see Section 6.6.10.2 [Vector Creation], page 187) can be modified using the following procedures.

**NOTE:** According to R5RS, it is an error to use any of these procedures on a literally read vector, because such vectors should be considered as constants. Currently, however, Guile does not detect this error.

**vector-set! vec k obj**  
[Scheme Procedure]  
Store obj in position k of vec. k must be a valid index of vec. The value returned by ‘vector-set!’ is unspecified.

(let ((vec (vector 0 '(2 2 2 2) "Anna")))  
(vector-set! vec 1 "Sue" "Sue")  
vec) ⇒ #(0 "Sue" "Sue") "Anna"

**void scm_c_vector_set_x (SCM vec, size_t k, SCM obj)**  
[C Function]  
Store obj in position k (a size_t) of vec.
vector-fill! vec fill

scm_vector_fill_x (vec, fill)

Store fill in every position of vec. The value returned by vector-fill! is unspecified.

vector-copy vec

scm_vector_copy (vec)

Return a copy of vec.

vector-move-left! vec1 start1 end1 vec2 start2

scm_vector_move_left_x (vec1, start1, end1, vec2, start2)

Copy elements from vec1, positions start1 to end1, to vec2 starting at position start2. start1 and start2 are inclusive indices; end1 is exclusive.

vector-move-left! copies elements in leftmost order. Therefore, in the case where vec1 and vec2 refer to the same vector, vector-move-left! is usually appropriate when start1 is greater than start2.

vector-move-right! vec1 start1 end1 vec2 start2

scm_vector_move_right_x (vec1, start1, end1, vec2, start2)

Copy elements from vec1, positions start1 to end1, to vec2 starting at position start2. start1 and start2 are inclusive indices; end1 is exclusive.

vector-move-right! copies elements in rightmost order. Therefore, in the case where vec1 and vec2 refer to the same vector, vector-move-right! is usually appropriate when start1 is less than start2.

6.6.10.4 Vector Accessing from C

A vector can be read and modified from C with the functions scm_c_vector_ref and scm_c_vector_set_x, for example. In addition to these functions, there are two more ways to access vectors from C that might be more efficient in certain situations: you can restrict yourself to simple vectors and then use the very fast simple vector macros; or you can use the very general framework for accessing all kinds of arrays (see Section 6.6.13.5 [Accessing Arrays from C], page 210), which is more verbose, but can deal efficiently with all kinds of vectors (and arrays). For vectors, you can use the scm_vector_elements and scm_vector_writable_elements functions as shortcuts.

int scm_is_simple_vector (SCM obj)

scm_c_vector_ref (SCM obj)

Return non-zero if obj is a simple vector, else return zero. A simple vector is a vector that can be used with the SCM_SIMPLE_* macros below.

The following functions are guaranteed to return simple vectors: scm_make_vector, scm_c_make_vector, scm_vector, scm_list_to_vector.

size_t SCM_SIMPLE_VECTOR_LENGTH (SCM vec)

scm_c_vector_length (SCM obj)

Evaluates to the length of the simple vector vec. No type checking is done.

SCM SCM_SIMPLE_VECTOR_REF (SCM vec, size_t idx)

scm_c_vector_ref (SCM obj)

Evaluates to the element at position idx in the simple vector vec. No type or range checking is done.

void SCM_SIMPLE_VECTOR_SET (SCM vec, size_t idx, SCM val)

scm_c_vector_set_x (SCM obj)

Sets the element at position idx in the simple vector vec to val. No type or range checking is done.
const SCM * scm_vector_elements (SCM vec, [C Function]
    scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
Acquire a handle for the vector vec and return a pointer to the elements of it. This
pointer can only be used to read the elements of vec. When vec is not a vector, an
error is signaled. The handle must eventually be released with scm_array_handle_release.

The variables pointed to by lenp and incp are filled with the number of elements
of the vector and the increment (number of elements) between successive elements,
respectively. Successive elements of vec need not be contiguous in their underlying
“root vector” returned here; hence the increment is not necessarily equal to 1 and
may well be negative too (see Section 6.6.13.3 [Shared Arrays], page 206).

The following example shows the typical way to use this function. It creates a list of
all elements of vec (in reverse order).

    scm_t_array_handle handle;
    size_t i, len;
    ssize_t inc;
    const SCM *elt;
    SCM list;

    elt = scm_vector_elements (vec, &handle, &len, &inc);
    list = SCM_EOL;
    for (i = 0; i < len; i++, elt += inc)
        list = scm_cons (*elt, list);
    scm_array_handle_release (&handle);

SCM * scm_vector_writable_elements (SCM vec, [C Function]
    scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
Like scm_vector_elements but the pointer can be used to modify the vector.

The following example shows the typical way to use this function. It fills a vector
with #t.

    scm_t_array_handle handle;
    size_t i, len;
    ssize_t inc;
    SCM *elt;

    elt = scm_vector_writable_elements (vec, &handle, &len, &inc);
    for (i = 0; i < len; i++, elt += inc)
        *elt = SCM_BOOL_T;
    scm_array_handle_release (&handle);

6.6.10.5 Uniform Numeric Vectors

A uniform numeric vector is a vector whose elements are all of a single numeric type. Guile
offers uniform numeric vectors for signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit
integers, two sizes of floating point values, and complex floating-point numbers of these two
sizes. See Section 7.5.5 [SRFI-4], page 592, for more information.
For many purposes, bytevectors work just as well as uniform vectors, and have the advantage that they integrate well with binary input and output. See Section 6.6.12 [Bytevectors], page 193, for more information on bytevectors.

6.6.11 Bit Vectors

Bit vectors are zero-origin, one-dimensional arrays of booleans. They are displayed as a sequence of 0s and 1s prefixed by **, e.g.,

\[
\text{(make-bitvector 8 #f) ⇒ }
\]
\[
**00000000
\]

Bit vectors are the special case of one dimensional bit arrays, and can thus be used with the array procedures, See Section 6.6.13 [Arrays], page 200.

\text{bitvector\? \ obj} \quad \text{[Scheme Procedure]}
\text{scm\_bitvector\_p (obj)} \quad \text{[C Function]}
\text{Return \#t when \ obj is a bitvector, else return \#f.}

\text{int \ scm\_is\_bitvector (SCM \ obj)} \quad \text{[C Function]}
\text{Return 1 when \ obj is a bitvector, else return 0.}

\text{make-bitvector \ len \ [fill]} \quad \text{[Scheme Procedure]}
\text{scm\_make\_bitvector (len, fill)} \quad \text{[C Function]}
\text{Create a new bitvector of length \ len and optionally initialize all elements to \ fill.}

\text{SCM \ scm\_c\_make\_bitvector (size\_t \ len, SCM \ fill)} \quad \text{[C Function]}
\text{Like \ scm\_make\_bitvector, but the length is given as a \size\_t.}

\text{bitvector \ bit ...} \quad \text{[Scheme Procedure]}
\text{scm\_bitvector (bits)} \quad \text{[C Function]}
\text{Create a new bitvector with the arguments as elements.}

\text{bitvector\-length \ vec} \quad \text{[Scheme Procedure]}
\text{scm\_bitvector\_length (vec)} \quad \text{[C Function]}
\text{Return the length of the bitvector \ vec.}

\text{size\_t \ scm\_c\_bitvector\_length (SCM \ vec)} \quad \text{[C Function]}
\text{Like \ scm\_bitvector\_length, but the length is returned as a \size\_t.}

\text{bitvector\-ref \ vec \ idx} \quad \text{[Scheme Procedure]}
\text{scm\_bitvector\_ref (vec, idx)} \quad \text{[C Function]}
\text{Return the element at index \ idx of the bitvector \ vec.}

\text{SCM \ scm\_c\_bitvector\_ref (SCM \ vec, size\_t \ idx)} \quad \text{[C Function]}
\text{Return the element at index \ idx of the bitvector \ vec.}

\text{bitvector\-set! \ vec \ idx \ val} \quad \text{[Scheme Procedure]}
\text{scm\_bitvector\_set\_x (vec, idx, val)} \quad \text{[C Function]}
\text{Set the element at index \ idx of the bitvector \ vec when \ val is true, else clear it.}

\text{SCM \ scm\_c\_bitvector\_set\_x (SCM \ vec, size\_t \ idx, SCM \ val)} \quad \text{[C Function]}
\text{Set the element at index \ idx of the bitvector \ vec when \ val is true, else clear it.}
bitvector-fill! vec val

Set all elements of the bitvector vec when val is true, else clear them.

list->bitvector list

Return a new bitvector initialized with the elements of list.

bitvector->list vec

Return a new list initialized with the elements of the bitvector vec.

bit-count bool bitvector

Return a count of how many entries in bitvector are equal to bool. For example,

(bit-count #f #*00011100) ⇒ 6

bit-position bool bitvector start

Return the index of the first occurrence of bool in bitvector, starting from start. If there is no bool entry between start and the end of bitvector, then return #f. For example,

(bit-position #t #*000101 0) ⇒ 3
(bit-position #f #*0001111 3) ⇒ #f

bit-invert! bitvector

Modify bitvector by replacing each element with its negation.

bit-set*! bitvector uvec bool

Set entries of bitvector to bool, with uvec selecting the entries to change. The return value is unspecified.

If uvec is a bit vector, then those entries where it has #t are the ones in bitvector which are set to bool. uvec and bitvector must be the same length. When bool is #t it’s like uvec is OR’ed into bitvector. Or when bool is #f it can be seen as an ANDNOT.

(define bv #*01000010)
(bit-set*! bv #*10010001 #t)
bv
⇒ #*11101001

If uvec is a uniform vector of unsigned long integers, then they’re indexes into bitvector which are set to bool.

(define bv #*01000010)
(bit-set*! bv #u(5 2 7) #t)
bv
⇒ #*01100111
Chapter 6: API Reference

**bit-count**  
* bitvector uvec bool  

[Scheme Procedure]

**scm_bit_count_star**  (bitvector, uvec, bool)  

Return a count of how many entries in *bitvector* are equal to *bool*, with *uvec* selecting the entries to consider.

*uvec* is interpreted in the same way as for *bit-set*! above. Namely, if *uvec* is a bit vector then entries which have #t there are considered in *bitvector*. Or if *uvec* is a uniform vector of unsigned long integers then it’s the indexes in *bitvector* to consider.

For example,

```
(bit-count#*01110111 #*11001101 #t) ⇒ 3
(bit-count#*01110111 #u32(7 0 4) #f) ⇒ 2
```

[C Function]

```
const scm_t_uint32 * scm_bitvector_elements (SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp)
```

Like *scm_vector_elements* (see Section 6.6.10.4 [Vector Accessing from C], page 189), but for bitvectors. The variable pointed to by *offp* is set to the value returned by *scm_array_handle_bit_elements_offset*. See *scm_array_handle_bit_elements* for how to use the returned pointer and the offset.

[C Function]

```
const scm_t_uint32 * scm_bitvector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp)
```

Like *scm_bitvector_elements*, but the pointer is good for reading and writing.

### 6.6.12 Bytevectors

A bytector is a raw bit string. The *(rnrs bytevectors)* module provides the programming interface specified by the Revised 6 Report on the Algorithmic Language Scheme (R6RS) ([http://www.r6rs.org/](http://www.r6rs.org/)). It contains procedures to manipulate bytevectors and interpret their contents in a number of ways: bytevector contents can be accessed as signed or unsigned integer of various sizes and endianness, as IEEE-754 floating point numbers, or as strings. It is a useful tool to encode and decode binary data.

The R6RS (Section 4.3.4) specifies an external representation for bytevectors, whereby the octets (integers in the range 0–255) contained in the bytevector are represented as a list prefixed by #vu8:

```
#vu8(1 53 204)
```

denotes a 3-byte bytevector containing the octets 1, 53, and 204. Like string literals, booleans, etc., bytevectors are “self-quoting”, i.e., they do not need to be quoted:

```
#vu8(1 53 204)
⇒ #vu8(1 53 204)
```

Bytevectors can be used with the binary input/output primitives (see Section 6.14.2 [Binary I/O], page 330).

### 6.6.12.1 Endianness

Some of the following procedures take an *endianness* parameter. The *endianness* is defined as the order of bytes in multi-byte numbers: numbers encoded in big endian have their
most significant bytes written first, whereas numbers encoded in little endian have their least significant bytes first\(^3\).

Little-endian is the native endianness of the IA32 architecture and its derivatives, while big-endian is native to SPARC and PowerPC, among others. The native-endianness procedure returns the native endianness of the machine it runs on.

```scheme
native-endianness
```

Returns a value denoting the native endianness of the host machine.

```scheme
endianness symbol
```

Returns an object denoting the endianness specified by `symbol`. If `symbol` is neither big nor little then an error is raised at expand-time.

```c
scm_endianness_big
scm_endianness_little
```

The objects denoting big- and little-endianness, respectively.

### 6.6.12.2 Manipulating Bytevectors

Bytevectors can be created, copied, and analyzed with the following procedures and C functions.

```scheme
make-bytevector len [fill]
```

 Returns a new bytevector of `len` bytes. Optionally, if `fill` is given, fill it with `fill`; `fill` must be in the range \([-128,255]\).  

```scheme
bytevector? obj
```

Returns true if `obj` is a bytevector.

```c
int scm_is_bytevector (SCM obj)
```

Equivalent to `scm_is_true (scm_bytevector_p (obj))`.

```scheme
bytevector-length bv
```

Returns the length in bytes of bytevector `bv`.

```c
size_t scm_c_bytevector_length (SCM bv)
```

Likewise, return the length in bytes of bytevector `bv`.

```scheme
bytevector=? bv1 bv2
```

Returns `bv1` equals to `bv2`—i.e., if they have the same length and contents.

---

\(^3\) Big-endian and little-endian are the most common “endianness”, but others do exist. For instance, the GNU MP library allows word order to be specified independently of byte order (see Section “Integer Import and Export” in *The GNU Multiple Precision Arithmetic Library Manual*).
Chapter 6: API Reference

`bytevector-fill! bv fill`  
[Scheme Procedure]  
Fill bytevector `bv` with `fill`, a byte.

`scm_bytevector_fill_x (bv, fill)`  
[C Function]  
Copy `len` bytes from `source` into `target`, starting reading from `source-start` (a positive index within `source`) and start writing at `target-start`. It is permitted for the `source` and `target` regions to overlap.

`bytevector-copy! source source-start target target-start len`  
[Scheme Procedure]  
Copy `len` bytes from `source` into `target`, starting reading from `source-start` (a positive index within `source`) and start writing at `target-start`. It is permitted for the `source` and `target` regions to overlap.

`bytevector-copy bv`  
[Scheme Procedure]  
Return a newly allocated copy of `bv`.

`scm_bytevector_copy (bv)`  
[C Function]  
Return a newly allocated copy of `bv`.

`scm_t_uint8 scm_c_bytevector_ref (SCM bv, size_t index)`  
[C Function]  
Return the byte at `index` in bytevector `bv`.

`void scm_c_bytevector_set_x (SCM bv, size_t index, scm_t_uint8 value)`  
[C Function]  
Set the byte at `index` in `bv` to `value`.

Low-level C macros are available. They do not perform any type-checking; as such they should be used with care.

`size_t SCM_BYTEVECTOR_LENGTH (bv)`  
[C Macro]  
Return the length in bytes of bytevector `bv`.

`signed char * SCM_BYTEVECTOR_CONTENTS (bv)`  
[C Macro]  
Return a pointer to the contents of bytevector `bv`.

### 6.6.12.3 Interpreting Bytevector Contents as Integers

The contents of a bytevector can be interpreted as a sequence of integers of any given size, sign, and endianness.

```scheme
(let ((bv (make-bytevector 4)))
  (bytevector-u8-set! bv 0 #x12)
  (bytevector-u8-set! bv 1 #x34)
  (bytevector-u8-set! bv 2 #x56)
  (bytevector-u8-set! bv 3 #x78)

  (map (lambda (number)
         (number->string number 16))
       (list (bytevector-u8-ref bv 0)
             (bytevector-u16-ref bv 0 (endianness big))
             (bytevector-u32-ref bv 0 (endianness little))))
 ⇒ ("12" "1234" "78563412")
)
```

The most generic procedures to interpret bytevector contents as integers are described below.
bytevector-uint-ref \ bv \ index \ endianness \ size

Return the size-byte long unsigned integer at index index in bv, decoded according to endianness.

bytevector-sint-ref \ bv \ index \ endianness \ size

Return the size-byte long signed integer at index index in bv, decoded according to endianness.

bytevector-uint-set! \ bv \ index \ value \ endianness \ size

Set the size-byte long unsigned integer at index to value, encoded according to endianness.

bytevector-sint-set! \ bv \ index \ value \ endianness \ size

Set the size-byte long signed integer at index to value, encoded according to endianness.

The following procedures are similar to the ones above, but specialized to a given integer size:

bytevector-u8-ref \ bv \ index

bytevector-s8-ref \ bv \ index

bytevector-u16-ref \ bv \ index \ endianness

bytevector-s16-ref \ bv \ index \ endianness

bytevector-u32-ref \ bv \ index \ endianness

bytevector-s32-ref \ bv \ index \ endianness

bytevector-u64-ref \ bv \ index \ endianness

bytevector-s64-ref \ bv \ index \ endianness

Return the unsigned \( n \)-bit (signed) integer (where \( n \) is 8, 16, 32 or 64) from \( \text{bv} \) at \( \text{index} \), decoded according to endianness.

bytevector-u8-set! \ bv \ index \ value

bytevector-s8-set! \ bv \ index \ value

bytevector-u16-set! \ bv \ index \ value \ endianness

bytevector-s16-set! \ bv \ index \ value \ endianness

bytevector-u32-set! \ bv \ index \ value \ endianness

bytevector-s32-set! \ bv \ index \ value \ endianness

bytevector-u64-set! \ bv \ index \ value \ endianness

bytevector-s64-set! \ bv \ index \ value \ endianness
Chapter 6: API Reference

bytevector-s64-set! bv index value endianness

[C Function]
scm_bytevector_u8_set_x (bv, index, value)
scm_bytevector_s8_set_x (bv, index, value)
scm_bytevector_u16_set_x (bv, index, value, endianness)
scm_bytevector_s16_set_x (bv, index, value, endianness)
scm_bytevector_u32_set_x (bv, index, value, endianness)
scm_bytevector_s32_set_x (bv, index, value, endianness)
scm_bytevector_u64_set_x (bv, index, value, endianness)
scm_bytevector_s64_set_x (bv, index, value, endianness)

Store value as an n-bit (signed) integer (where n is 8, 16, 32 or 64) in bv at index, encoded according to endianness.

Finally, a variant specialized for the host’s endianness is available for each of these functions (with the exception of the u8 and s8 accessors, as endianness is about byte order and there is only 1 byte):

bytevector-u16-native-ref bv index
bytevector-s16-native-ref bv index
bytevector-u32-native-ref bv index
bytevector-s32-native-ref bv index
bytevector-u64-native-ref bv index
bytevector-s64-native-ref bv index

[C Function]
scm_bytevector_u16_native_ref (bv, index)
scm_bytevector_s16_native_ref (bv, index)
scm_bytevector_u32_native_ref (bv, index)
scm_bytevector_s32_native_ref (bv, index)
scm_bytevector_u64_native_ref (bv, index)
scm_bytevector_s64_native_ref (bv, index)

Return the unsigned n-bit (signed) integer (where n is 8, 16, 32 or 64) from bv at index, decoded according to the host’s native endianness.

bytevector-u16-native-set! bv index value
bytevector-s16-native-set! bv index value
bytevector-u32-native-set! bv index value
bytevector-s32-native-set! bv index value
bytevector-u64-native-set! bv index value
bytevector-s64-native-set! bv index value

[C Function]
scm_bytevector_u16_native_set_x (bv, index, value)
scm_bytevector_s16_native_set_x (bv, index, value)
scm_bytevector_u32_native_set_x (bv, index, value)
scm_bytevector_s32_native_set_x (bv, index, value)
scm_bytevector_u64_native_set_x (bv, index, value)
scm_bytevector_s64_native_set_x (bv, index, value)

Store value as an n-bit (signed) integer (where n is 8, 16, 32 or 64) in bv at index, encoded according to the host’s native endianness.

6.6.12.4 Converting Bytevectors to/from Integer Lists

Bytevector contents can readily be converted to/from lists of signed or unsigned integers:
(bytevector->sint-list (u8-list->bytevector (make-list 4 255))
  (endianness little) 2)
⇒ (-1 -1)

(bytevector->u8-list bv)  [Scheme Procedure]
(scm_bytevector_to_u8_list (bv))  [C Function]
   Return a newly allocated list of unsigned 8-bit integers from the contents of bv.

(u8-list->bytevector lst)  [Scheme Procedure]
(scm_u8_list_to_bytevector (lst))  [C Function]
   Return a newly allocated bytevector consisting of the unsigned 8-bit integers listed in lst.

(bytevector->uint-list bv endianness size)  [Scheme Procedure]
(scm_bytevector_to_uint_list (bv, endianness, size))  [C Function]
   Return a list of unsigned integers of size bytes representing the contents of bv, decoded according to endianness.

(bytevector->sint-list bv endianness size)  [Scheme Procedure]
(scm_bytevector_to_sint_list (bv, endianness, size))  [C Function]
   Return a list of signed integers of size bytes representing the contents of bv, decoded according to endianness.

(uint-list->bytevector lst endianness size)  [Scheme Procedure]
(scm_uint_list_to_bytevector (lst, endianness, size))  [C Function]
   Return a new bytevector containing the unsigned integers listed in lst and encoded on size bytes according to endianness.

(sint-list->bytevector lst endianness size)  [Scheme Procedure]
(scm_sint_list_to_bytevector (lst, endianness, size))  [C Function]
   Return a new bytevector containing the signed integers listed in lst and encoded on size bytes according to endianness.

### 6.6.12.5 Interpreting Bytevector Contents as Floating Point Numbers

Bytevector contents can also be accessed as IEEE-754 single- or double-precision floating point numbers (respectively 32 and 64-bit long) using the procedures described here.

(bytevector-ieee-single-ref bv index endianness)  [Scheme Procedure]
(bytevector-ieee-double-ref bv index endianness)  [Scheme Procedure]
(scm_bytevector_ieee_single_ref (bv, index, endianness))  [C Function]
(scm_bytevector_ieee_double_ref (bv, index, endianness))  [C Function]
   Return the IEEE-754 single-precision floating point number from bv at index according to endianness.

(bytevector-ieee-single-set! bv index value endianness)  [Scheme Procedure]
(bytevector-ieee-double-set! bv index value endianness)  [Scheme Procedure]
(scm_bytevector_ieee_single_set_x (bv, index, value, endianness))  [C Function]
`scm_bytevector_ieee_double_set_x` (bv, index, value, endianness)

Store real number value in bv at index according to endianness.

Specialized procedures are also available:

- `bytevector-ieee-single-native-ref` bv index
- `bytevector-ieee-double-native-ref` bv index

```
(scm_bytevector_ieee_single_native_ref bv index)
```

Return the IEEE-754 single-precision floating point number from bv at index according to the host’s native endianness.

- `bytevector-ieee-single-native-set!` bv index value
- `bytevector-ieee-double-native-set!` bv index value

```
(scm_bytevector_ieee_double_native_set_x bv index value)
```

Store real number value in bv at index according to the host’s native endianness.

### 6.6.12.6 Interpreting Bytevector Contents as Unicode Strings

Bytevector contents can also be interpreted as Unicode strings encoded in one of the most commonly available encoding formats. See Section 6.6.5.13 [Representing Strings as Bytes], page 159, for a more generic interface.

```
(utf8->string (u8-list->bytevector '(99 97 102 101)))
⇒ "cafe"
```

```
(string->utf8 "café") ;; SMALL LATIN LETTER E WITH ACUTE ACCENT
⇒ #vu8(99 97 102 195 169)
```

- `string-utf8-length` str
- `size_t scm_c_string_utf8_length` (str)

```
(scm_c_string_utf8_length str)
```

Return the number of bytes in the UTF-8 representation of str.

- `string->utf8` str
- `string->utf16` str [endianness]
- `string->utf32` str [endianness]
- `scm_string_to_utf8` (str)
- `scm_string_to_utf16` (str, endianness)
- `scm_string_to_utf32` (str, endianness)

```
(scm_string_to_utf32 str endianness)
```

Return a newly allocated bytevector that contains the UTF-8, UTF-16, or UTF-32 (aka. UCS-4) encoding of str. For UTF-16 and UTF-32, endianness should be the symbol `big` or `little`; when omitted, it defaults to big endian.

- `utf8->string` utf
- `utf16->string` utf [endianness]
- `utf32->string` utf [endianness]

```
(scm_utf8_to_string utf)
```

[C Function]
Return a newly allocated string that contains from the UTF-8-, UTF-16-, or UTF-32-decoded contents of bytevector `utf`. For UTF-16 and UTF-32, `endianness` should be the symbol `big` or `little`; when omitted, it defaults to big endian.

### 6.6.12.7 Accessing Bytevectors with the Array API
As an extension to the R6RS, Guile allows bytevectors to be manipulated with the `array` procedures (see Section 6.6.13 [Arrays], page 200). When using these APIs, bytes are accessed one at a time as 8-bit unsigned integers:

```scheme
(define bv #vu8(0 1 2 3))
```

```
(array? bv)
⇒ #t

(array-rank bv)
⇒ 1

(array-ref bv 2)
⇒ 2

;; Note the different argument order on array-set!.
(array-set! bv 77 2)

(array-ref bv 2)
⇒ 77
```

```
(array-type bv)
⇒ vu8
```

### 6.6.12.8 Accessing Bytevectors with the SRFI-4 API
Bytevectors may also be accessed with the SRFI-4 API. See Section 7.5.5.3 [SRFI-4 and Bytevectors], page 600, for more information.

### 6.6.13 Arrays
Arrays are a collection of cells organized into an arbitrary number of dimensions. Each cell can be accessed in constant time by supplying an index for each dimension.

In the current implementation, an array uses a vector of some kind for the actual storage of its elements. Any kind of vector will do, so you can have arrays of uniform numeric values, arrays of characters, arrays of bits, and of course, arrays of arbitrary Scheme values. For example, arrays with an underlying `c64vector` might be nice for digital signal processing, while arrays made from a `u8vector` might be used to hold gray-scale images.

The number of dimensions of an array is called its `rank`. Thus, a matrix is an array of rank 2, while a vector has rank 1. When accessing an array element, you have to specify one exact integer for each dimension. These integers are called the `indices` of the element. An array specifies the allowed range of indices for each dimension via an inclusive lower and upper bound. These bounds can well be negative, but the upper bound must be greater
than or equal to the lower bound minus one. When all lower bounds of an array are zero, it is called a zero-origin array.

Arrays can be of rank 0, which could be interpreted as a scalar. Thus, a zero-rank array can store exactly one object and the list of indices of this element is the empty list.

Arrays contain zero elements when one of their dimensions has a zero length. These empty arrays maintain information about their shape: a matrix with zero columns and 3 rows is different from a matrix with 3 columns and zero rows, which again is different from a vector of length zero.

The array procedures are all polymorphic, treating strings, uniform numeric vectors, bytevectors, bit vectors and ordinary vectors as one dimensional arrays.

### 6.6.13.1 Array Syntax

An array is displayed as `#` followed by its rank, followed by a tag that describes the underlying vector, optionally followed by information about its shape, and finally followed by the cells, organized into dimensions using parentheses.

In more words, the array tag is of the form

```
#<rank><vectag><@lower><:len><@lower><:len>...
```

where `<rank>` is a positive integer in decimal giving the rank of the array. It is omitted when the rank is 1 and the array is non-shared and has zero-origin (see below). For shared arrays and for a non-zero origin, the rank is always printed even when it is 1 to distinguish them from ordinary vectors.

The `<vectag>` part is the tag for a uniform numeric vector, like `u8`, `s16`, etc, `b` for bitvectors, or `a` for strings. It is empty for ordinary vectors.

The `<@lower>` part is a `'@'` character followed by a signed integer in decimal giving the lower bound of a dimension. There is one `<@lower>` for each dimension. When all lower bounds are zero, all `<@lower>` parts are omitted.

The `<:len>` part is a `':` character followed by an unsigned integer in decimal giving the length of a dimension. Like for the lower bounds, there is one `<:len>` for each dimension, and the `<:len>` part always follows the `<@lower>` part for a dimension. Lengths are only then printed when they can’t be deduced from the nested lists of elements of the array literal, which can happen when at least one length is zero.

As a special case, an array of rank 0 is printed as `#0<vectag>(<scalar>)`, where `<scalar>` is the result of printing the single element of the array.

Thus,

```
#(1 2 3)
```

is an ordinary array of rank 1 with lower bound 0 in dimension 0. (I.e., a regular vector.)

```
#@2(1 2 3)
```

is an ordinary array of rank 1 with lower bound 2 in dimension 0.

```
#2((1 2 3) (4 5 6))
```

is a non-uniform array of rank 2; a 2×3 matrix with index ranges 0..1 and 0..2.

```
#u8(0 1 2)
```

is a uniform u8 array of rank 1.
is a uniform u32 array of rank 2 with index ranges 2..3 and 3..4.

is a two-dimensional array with index ranges 0..-1 and 0..-1, i.e. both dimensions have length zero.

is a two-dimensional array with index ranges 0..-1 and 0..1, i.e. the first dimension has length zero, but the second has length 2.

is a rank-zero array with contents 12.

In addition, bytevectors are also arrays, but use a different syntax (see Section 6.6.12 [Bytevectors], page 193):

is a 3-byte long bytevector, with contents 1, 2, 3.

### 6.6.13.2 Array Procedures

When an array is created, the range of each dimension must be specified, e.g., to create a 2×3 array with a zero-based index:

```
(make-array 'ho 2 3) ⇒ #2((ho ho ho) (ho ho ho))
```

The range of each dimension can also be given explicitly, e.g., another way to create the same array:

```
(make-array 'ho '(0 1) '(0 2)) ⇒ #2((ho ho ho) (ho ho ho))
```

The following procedures can be used with arrays (or vectors). An argument shown as `idx...` means one parameter for each dimension in the array. A `idxlist` argument means a list of such values, one for each dimension.

```
array? obj [Scheme Procedure]
scm_array_p (obj, unused) [C Function]
  Return #t if the obj is an array, and #f if not.
  The second argument to scm_array_p is there for historical reasons, but it is not used.
  You should always pass SCM_UNDEFINED as its value.

typed-array? obj type [Scheme Procedure]
scm_typed_array_p (obj, type) [C Function]
  Return #t if the obj is an array of type type, and #f if not.

int scm_is_array (SCM obj) [C Function]
  Return 1 if the obj is an array and 0 if not.

int scm_is_typed_array (SCM obj, SCM type) [C Function]
  Return 0 if the obj is an array of type type, and 1 if not.

make-array fill bound ... [Scheme Procedure]
scm_make_array (fill, bounds) [C Function]
  Equivalent to (make-typed-array #t fill bound ...).
```
make-typed-array type fill bound ...  

**Scheme Procedure**

**scm_make_typed_array** (type, fill, bounds)

Create and return an array that has as many dimensions as there are `bounds` and (maybe) fill it with `fill`.

The underlying storage vector is created according to `type`, which must be a symbol whose name is the ‘vectag’ of the array as explained above, or `#t` for ordinary, non-specialized arrays.

For example, using the symbol `f64` for `type` will create an array that uses a `f64vector` for storing its elements, and `a` will use a string.

When `fill` is not the special `unspecified` value, the new array is filled with `fill`. Otherwise, the initial contents of the array is unspecified. The special `unspecified` value is stored in the variable `*unspecified*` so that for example `(make-typed-array 'u32 *unspecified* 4)` creates a uninitialized `u32` vector of length 4.

Each `bound` may be a positive non-zero integer `n`, in which case the index for that dimension can range from 0 through `n-1`; or an explicit index range specifier in the form `(LOWER UPPER)`, where both `lower` and `upper` are integers, possibly less than zero, and possibly the same number (however, `lower` cannot be greater than `upper`).

**list->array dimspec list**  

**Scheme Procedure**

Equivalent to `(list->typed-array #t dimspec list).`

**list->typed-array type dimspec list**  

**Scheme Procedure**

**scm_list_to_typed_array** (type, dimspec, list)

**C Function**

Return an array of the type indicated by `type` with elements the same as those of `list`.

The argument `dimspec` determines the number of dimensions of the array and their lower bounds. When `dimspec` is an exact integer, it gives the number of dimensions directly and all lower bounds are zero. When it is a list of exact integers, then each element is the lower index bound of a dimension, and there will be as many dimensions as elements in the list.

**array-type array**  

**Scheme Procedure**

**scm_array_type** (array)

**C Function**

Return the type of `array`. This is the ‘vectag’ used for printing `array` (or `#t` for ordinary arrays) and can be used with `make-typed-array` to create an array of the same kind as `array`.

**array-ref array idx ...**  

**Scheme Procedure**

**scm_array_ref** (array, idxlist)

**C Function**

Return the element at `(idx ...)` in `array`.

```
(define a (make-array 999 '(1 2) '(3 4)))
(array-ref a 2 4) ⇒ 999
```

**array-in-bounds? array idx ...**  

**Scheme Procedure**

**scm_array_in_bounds_p** (array, idxlist)

**C Function**

Return `#t` if the given indices would be acceptable to `array-ref`.

```
(define a (make-array #f '(1 2) '(3 4)))
(array-in-bounds? a 2 3) ⇒ #t
(array-in-bounds? a 0 0) ⇒ #f
```
array-set!  array  obj  idx . . .  

scm_array_set_x (array, obj, idxlist)

Set the element at (idx . . . ) in array to obj. The return value is unspecified.

(define a (make-array #f '(0 1) '(0 1)))
(array-set! a #t 1 1)
a  ⇒  #2((#f #f) (#f #t))

array-shape  array

array-dimensions  array

scm_array_dimensions (array)

Return a list of the bounds for each dimension of array. array-shape gives (lower upper) for each dimension. array-dimensions instead returns just upper + 1 for dimensions with a 0 lower bound. Both are suitable as input to make-array.

For example,

(define a (make-array 'foo '(-1 3) 5))
(array-shape a)  ⇒  ((-1 3) (0 4))
(array-dimensions a)  ⇒  ((-1 3) 5)

array-length  array

scm_array_length (array)

size_t  scm_c_array_length (array)

Return the length of an array: its first dimension. It is an error to ask for the length of an array of rank 0.

array-rank  array

scm_array_rank (array)

size_t  scm_c_array_rank (SCM array)

Return the rank of array as a size_t.

array->list  array

scm_array_to_list (array)

Return a list consisting of all the elements, in order, of array.

array-copy!  src  dst

array-copy-in-order!  src  dst

scm_array_copy_x (src, dst)

Copy every element from vector or array src to the corresponding element of dst. dst must have the same rank as src, and be at least as large in each dimension. The return value is unspecified.

array-fill!  array  fill

scm_array_fill_x (array, fill)

Store fill in every element of array. The value returned is unspecified.
array-equal?  array . . .

Return #t if all arguments are arrays with the same shape, the same type, and have
 corresponding elements which are either equal? or array-equal?. This function
differs from equal? (see Section 6.11.1 [Equality], page 283) in that all arguments
must be arrays.

array-map!  dst proc src . . .

array-map-in-order! dst proc src . . .

scm_array_map_x (dst, proc, srclist)

Set each element of the dst array to values obtained from calls to proc. The list of
src arguments may be empty. The value returned is unspecified.

Each call is (proc elem . . .), where each elem is from the corresponding src array, at
the dst index. array-map-in-order! makes the calls in row-major order, array-map!
makes them in an unspecified order.

The src arrays must have the same number of dimensions as dst, and must have a
range for each dimension which covers the range in dst. This ensures all dst indices
are valid in each src.

array-for-each  proc src1 src2 . . .

scm_array_for_each (proc, src1, srclist)

Apply proc to each tuple of elements of src1 src2 . . ., in row-major order. The value
returned is unspecified.

array-index-map!  dst proc

scm_array_index_map_x (dst, proc)

Set each element of the dst array to values returned by calls to proc. The value
returned is unspecified.

Each call is (proc i1 . . . iN), where i1 . . . iN is the destination index, one parameter
for each dimension. The order in which the calls are made is unspecified.

For example, to create a 4 × 4 matrix representing a cyclic group,

\[
\begin{pmatrix}
0 & 1 & 2 & 3 \\
1 & 2 & 3 & 0 \\
2 & 3 & 0 & 1 \\
3 & 0 & 1 & 2
\end{pmatrix}
\]

(define a (make-array #f 4 4))
(array-index-map! a (lambda (i j)
  (modulo (+ i j) 4)))

An additional array function is available in the module (ice-9 arrays). It can be used
with:

(use-modules (ice-9 arrays))

array-copy src

Return a new array with the same elements, type and shape as src. However, the
array increments may not be the same as those of src. In the current implementation,
the returned array will be in row-major order, but that might change in the future. Use array-copy! on an array of known order if that is a concern.
### 6.6.13.3 Shared Arrays

**make-shared-array** *oldarray mapfunc bound ...*  

**Scheme Procedure**

**scm_make_shared_array** *(oldarray, mapfunc, boundlist)*  

**C Function**

Return a new array which shares the storage of *oldarray*. Changes made through either affect the same underlying storage. The *bound ...* arguments are the shape of the new array, the same as **make-array** (see Section 6.6.13.2 [Array Procedures], page 202).

*mapfunc* translates coordinates from the new array to the *oldarray*. It’s called as **(mapfunc newidx1 ...)** with one parameter for each dimension of the new array, and should return a list of indices for *oldarray*, one for each dimension of *oldarray*.

*mapfunc* must be affine linear, meaning that each *oldarray* index must be formed by adding integer multiples (possibly negative) of some or all of *newidx1* etc, plus a possible integer offset. The multiples and offset must be the same in each call.

One good use for a shared array is to restrict the range of some dimensions, so as to apply say **array-for-each** or **array-fill!** to only part of an array. The plain **list** function can be used for **mapfunc** in this case, making no changes to the index values. For example,

**(make-shared-array #2((a b c) (d e f) (g h i)) list 3 2)**  

⇒ #2((a b) (d e) (g h))

The new array can have fewer dimensions than *oldarray*, for example to take a column from an array.

**(make-shared-array #2((a b c) (d e f) (g h i))**  

**lambda** *(i) (list i 2))**  

'*(0 2))**

⇒ #1(c f i)

A diagonal can be taken by using the single new array index for both row and column in the old array. For example,

**(make-shared-array #2((a b c) (d e f) (g h i))**  

**lambda** *(i) (list i i))**  

'*(0 2))**

⇒ #1(a e i)

Dimensions can be increased by for instance considering portions of a one dimensional array as rows in a two dimensional array. (**array-contents** below can do the opposite, flattening an array.)

**(make-shared-array #1(a b c d e f g h i j k l)**  

**lambda** *(i j) (list (+ (* i 3) j)))**  

4 3)**

⇒ #2((a b c) (d e f) (g h i) (j k l))

By negating an index the order that elements appear can be reversed. The following just reverses the column order,

**(make-shared-array #2((a b c) (d e f) (g h i))**  

**lambda** *(i j) (list i (- 2 j)))**
A fixed offset on indexes allows for instance a change from a 0 based to a 1 based array,

\[
\begin{align*}
\text{(define } x &\text{ #2((a b c) (d e f) (g h i)))} \\
\text{(define } y &\text{ (make-shared-array } x \\
&\text{ (lambda (i j) (list (1- i) (1- j))))} \\
&\text{ '(1 3) '}(1 3)))
\end{align*}
\]

This can be extended for multi-dimensional arrays.

\[
\begin{align*}
\text{(array-ref } x &\text{ 0 0)} \Rightarrow a \\
\text{(array-ref } y &\text{ 1 1)} \Rightarrow a
\end{align*}
\]

A multiple on an index allows every Nth element of an array to be taken. The following is every third element,

\[
\begin{align*}
\text{(make-shared-array } #1(a b c d e f g h i j k l) \\
&\text{ (lambda (i) (list (* i 3)))}
\end{align*}
\]

\[
\Rightarrow #1(a d g j)
\]

The above examples can be combined to make weird and wonderful selections from an array, but it’s important to note that because mapfunc must be affine linear, arbitrary permutations are not possible.

In the current implementation, mapfunc is not called for every access to the new array but only on some sample points to establish a base and stride for new array indices in oldarray data. A few sample points are enough because mapfunc is linear.

**shared-array-increments** array

For each dimension, return the distance between elements in the root vector.

**shared-array-offset** array

Return the root vector index of the first element in the array.

**shared-array-root** array

Return the root vector of a shared array.

**array-contents** array [strict]

If array may be unrolled into a one dimensional shared array without changing their order (last subscript changing fastest), then array-contents returns that shared array, otherwise it returns #f. All arrays made by make-array and make-typed-array may be unrolled, some arrays made by make-shared-array may not be.

If the optional argument strict is provided, a shared array will be returned only if its elements are stored internally contiguous in memory.

**transpose-array** array dim1 dim2 ...

Return an array sharing contents with array, but with dimensions arranged in a different order. There must be one dim argument for each dimension of array. dim1,
\[ \text{dim2, \ldots should be integers between 0 and the rank of the array to be returned. Each integer in that range must appear at least once in the argument list.} \]

The values of \text{dim1}, \text{dim2}, \ldots correspond to dimensions in the array to be returned, and their positions in the argument list to dimensions of \text{array}. Several \text{dms} may have the same value, in which case the returned array will have smaller rank than \text{array}.

\[
\begin{align*}
\text{transpose-array'} &\#2((a \ b) \ (c \ d)) 1 0 \Rightarrow \#2((a \ c) \ (b \ d)) \\
\text{transpose-array'} &\#2((a \ b) \ (c \ d)) 0 0 \Rightarrow \#1(a \ d) \\
\text{transpose-array'} &\#3(((a \ b \ c) \ (d \ e \ f)) ((1 \ 2 \ 3) \ (4 \ 5 \ 6))) 1 1 0 \Rightarrow \\
&\#2((a \ 4) \ (b \ 5) \ (c \ 6))
\end{align*}
\]

### 6.6.13.4 Arrays as arrays of arrays

Mathematically, one can see an array of rank \(n\) (an \(n\)-array) as an array of lower rank where the elements are themselves arrays ('cells').

We speak of the first \(n-k\) dimensions of the array as the \(n-k\)-'frame' of the array, while the last \(k\) dimensions are the dimensions of the \(k\)-'cells'. For example, a 3-array can be seen as a 2-array of vectors (1-arrays) or as a 1-array of matrices (2-arrays). In each case, the vectors or matrices are the 1-cells or 2-cells of the array. This terminology originates in the J language.

The more vague concept of a 'slice' refers to a subset of the array where some indices are fixed and others are left free. As a Guile data object, a cell is the same as a 'prefix slice' (the first \(n-k\) indices into the original array are fixed), except that a 0-cell is not a shared array of the original array, but a 0-slice (where all the indices into the original array are fixed) is.

Before version 2.0, Guile had a feature called 'enclosed arrays' to create special 'array of arrays' objects. The functions in this section do not need special types; instead, the frame rank is stated in each function call, either implicitly or explicitly.

**array-cell-ref array idx ...**

**scm_array_cell_ref (array, idxlist)**

If the length of \text{idxlist} equals the rank \(n\) of \text{array}, return the element at \((\text{idx} \ldots)\), just like \text{(array-ref array idx ...)}. If, however, the length \(k\) of \text{idxlist} is smaller than \(n\), then return the \((n-k)\)-cell of \text{array} given by \text{idxlist}, as a shared array.

For example:

\[
\begin{align*}
\text{array-cell-ref'} &\#2((a \ b) \ (c \ d)) 0 \Rightarrow \#(a \ b) \\
\text{array-cell-ref'} &\#2((a \ b) \ (c \ d)) 1 \Rightarrow \#(c \ d) \\
\text{array-cell-ref'} &\#2((a \ b) \ (c \ d)) 1 \ 1 \ \Rightarrow \, d \\
\text{array-cell-ref'} &\#2((a \ b) \ (c \ d)) \Rightarrow \#2((a \ b) \ (c \ d))
\end{align*}
\]

\(\text{(apply array-cell-ref array indices)}\) is equivalent to

\[
\begin{align*}
\text{(let ((len (length indices)))} \\
\text{(if (= (array-rank a) len) } \\
\text{(apply array-ref a indices) } \\
\text{(apply make-shared-array a } \\
\text{(lambda t (append indices t))} \\
\text{(drop (array-dimensions a len))))})
\end{align*}
\]
array-slice array idx ...  

C Function
scm_array_slice (array, idxlist)

Like (array-cell-ref array idx ...), but return a 0-rank shared array into ARRAY if the length of idxlist matches the rank of array. This can be useful when using ARRAY as a place to write to.

Compare:

(array-cell-ref #2((a b) (c d)) 1 1) ⇒ d
(array-slice #2((a b) (c d)) 1 1) ⇒ #0(d)
(define a (make-array 'a 2 2))
(array-fill! (array-slice a 1 1) 'b)
a ⇒ #2((a a) (a b)).
(array-fill! (array-cell-ref a 1 1) 'b) ⇒ error: not an array

(apply array-slice array indices) is equivalent to

(apply make-shared-array a
  (lambda t (append indices t))
  (drop (array-dimensions a) (length indices)))

array-cell-set! array x idx ...  

C Function
scm_array_cell_set_x (array, x, idxlist)

If the length of idxlist equals the rank n of array, set the element at (idx ...) of array to x, just like (array-set! array x idx ...). If, however, the length k of idxlist is smaller than n, then copy the (n − k)-rank array x into the (n − k)-cell of array given by idxlist. In this case, the last (n − k) dimensions of array and the dimensions of x must match exactly.

This function returns the modified array.

For example:

(array-cell-set! (make-array 'a 2 2) b 1 1)
⇒ #2((a a) (a b))
(array-cell-set! (make-array 'a 2 2) #(x y) 1)
⇒ #2((a a) (x y))

Note that array-cell-set! will expect elements, not arrays, when the destination has rank 0. Use array-slice for the opposite behavior.

(array-cell-set! (make-array 'a 2 2) #0(b) 1 1)
⇒ #2((a a) (a #0(b)))
(let ((a (make-array 'a 2 2)))
  (array-copy! #0(b) (array-slice a 1 1)) a)
⇒ #2((a a) (a b))

(apply array-cell-set! array x indices) is equivalent to

(let ((len (length indices)))
  (if (= (array-rank array) len)
    (apply array-set! array x indices)
    (array-copy! x (apply array-cell-ref array indices)))
  array)
array-slice-for-each  frame-rank  op  x . . .  

Each x must be an array of rank \( \geq \) frame-rank, and the first frame-rank dimensions of each x must all be the same. array-slice-for-each calls op with each set of (rank(x) - frame-rank)-cells from x, in unspecified order.

array-slice-for-each allows you to loop over cells of any rank without having to carry an index list or construct shared arrays manually. The slices passed to op are always shared arrays of X, even if they are of rank 0, so it is possible to write to them.

This function returns an unspecified value.

For example, to sort the rows of rank-2 array a:

\[
\text{array-slice-for-each 1 (lambda (x) (sort! x <)) a)}
\]

As another example, let a be a rank-2 array where each row is a 2-element vector \((x, y)\). Let’s compute the arguments of these vectors and store them in rank-1 array b.

\[
\text{(array-slice-for-each 1 (lambda (a b) (array-set! b (atan (array-ref a 1) (array-ref a 0))) a b))}
\]

(array-slice-for-each frame-rank op x) is equivalent to

\[
\text{(let ((frame (take (array-dimensions (car x)) frank))) (unless (every (lambda (x) (equal? frame (take (array-dimensions x) frank))) (cdr x)))
(error))}
\]

(array-index-map!
(apply make-shared-array (make-array @t) (const '()) frame)
(lambda i (apply op (map (lambda (x) (apply array-slice x i)) x)))))

6.6.13.5 Accessing Arrays from C

For interworking with external C code, Guile provides an API to allow C code to access the elements of a Scheme array. In particular, for uniform numeric arrays, the API exposes the underlying uniform data as a C array of numbers of the relevant type.

While pointers to the elements of an array are in use, the array itself must be protected so that the pointer remains valid. Such a protected array is said to be reserved. A reserved array can be read but modifications to it that would cause the pointer to its elements to become invalid are prevented. When you attempt such a modification, an error is signalled.

(This is similar to locking the array while it is in use, but without the danger of a deadlock. In a multi-threaded program, you will need additional synchronization to avoid modifying reserved arrays.)
You must take care to always unreserve an array after reserving it, even in the presence of non-local exits. If a non-local exit can happen between these two calls, you should install a dynwind context that releases the array when it is left (see Section 6.13.10 [Dynamic Wind], page 317).

In addition, array reserving and unreserving must be properly paired. For instance, when reserving two or more arrays in a certain order, you need to unreserve them in the opposite order.

Once you have reserved an array and have retrieved the pointer to its elements, you must figure out the layout of the elements in memory. Guile allows slices to be taken out of arrays without actually making a copy, such as making an alias for the diagonal of a matrix that can be treated as a vector. Arrays that result from such an operation are not stored contiguously in memory and when working with their elements directly, you need to take this into account.

The layout of array elements in memory can be defined via a mapping function that computes a scalar position from a vector of indices. The scalar position then is the offset of the element with the given indices from the start of the storage block of the array.

In Guile, this mapping function is restricted to be affine: all mapping functions of Guile arrays can be written as $p = b + c[0]*i[0] + c[1]*i[1] + \ldots + c[n-1]*i[n-1]$ where $i[k]$ is the $k$th index and $n$ is the rank of the array. For example, a matrix of size $3\times3$ would have $b == 0$, $c[0] == 3$ and $c[1] == 1$. When you transpose this matrix (with `transpose-array`, say), you will get an array whose mapping function has $b == 0$, $c[0] == 1$ and $c[1] == 3$.

The function `scm_array_handle_dims` gives you (indirect) access to the coefficients $c[k]$.

Note that there are no functions for accessing the elements of a character array yet. Once the string implementation of Guile has been changed to use Unicode, we will provide them.

### scm_t_array_handle

This is a structure type that holds all information necessary to manage the reservation of arrays as explained above. Structures of this type must be allocated on the stack and must only be accessed by the functions listed below.

#### void scm_array_get_handle (SCM array, scm_t_array_handle *handle)

Reserve `array`, which must be an array, and prepare `handle` to be used with the functions below. You must eventually call `scm_array_handle_release` on `handle`, and do this in a properly nested fashion, as explained above. The structure pointed to by `handle` does not need to be initialized before calling this function.

#### void scm_array_handle_release (scm_t_array_handle *handle)

End the array reservation represented by `handle`. After a call to this function, `handle` might be used for another reservation.

#### size_t scm_array_handle_rank (scm_t_array_handle *handle)

Return the rank of the array represented by `handle`. 
### scm_t_array_dim

This structure type holds information about the layout of one dimension of an array. It includes the following fields:

- **ssize_t lbnd**
- **ssize_t ubnd**
  
  The lower and upper bounds (both inclusive) of the permissible index range for the given dimension. Both values can be negative, but `lbnd` is always less than or equal to `ubnd`.

- **ssize_t inc**
  
  The distance from one element of this dimension to the next. Note, too, that this can be negative.

### const scm_t_array_dim * scm_array_handle_dims

This C function returns a pointer to a C vector of information about the dimensions of the array represented by `handle`. This pointer is valid as long as the array remains reserved. As explained above, the `scm_t_array_dim` structures returned by this function can be used to calculate the position of an element in the storage block of the array from its indices.

This position can then be used as an index into the C array pointer returned by the various `scm_array_handle_<foo>_elements` functions, or with `scm_array_handle_ref` and `scm_array_handle_set`.

Here is how one can compute the position `pos` of an element given its indices in the vector `indices`:

```c
ssize_t indices[RANK];
scm_t_array_dim *dims;
ssize_t pos;
size_t i;

pos = 0;
for (i = 0; i < RANK; i++)
{
    if (indices[i] < dims[i].lbnd || indices[i] > dims[i].ubnd)
        out_of_range();
    pos += (indices[i] - dims[i].lbnd) * dims[i].inc;
}
```

### ssize_t scm_array_handle_pos

This C function computes the position corresponding to `indices`, a list of indices. The position is computed as described above for `scm_array_handle_dims`. The number of the indices and their range is checked and an appropriate error is signalled for invalid indices.

### SCM scm_array_handle_ref

This C function returns the element at position `pos` in the storage block of the array represented by `handle`. Any kind of array is acceptable. No range checking is done on `pos`. 
void scm_array_handle_set (scm_t_array_handle *handle, ssize_t pos, SCM val) [C Function]
    Set the element at position pos in the storage block of the array represented by handle to val. Any kind of array is acceptable. No range checking is done on pos. An error is signalled when the array can not store val.

const SCM * scm_array_handle_elements (scm_t_array_handle *handle) [C Function]
    Return a pointer to the elements of a ordinary array of general Scheme values (i.e., a non-uniform array) for reading. This pointer is valid as long as the array remains reserved.

SCM * scm_array_handle_writable_elements (scm_t_array_handle *handle) [C Function]
    Like scm_array_handle_elements, but the pointer is good for reading and writing.

const void * scm_array_handle_uniform_elements (scm_t_array_handle *handle) [C Function]
    Return a pointer to the elements of a uniform numeric array for reading. This pointer is valid as long as the array remains reserved. The size of each element is given by scm_array_handle_uniform_element_size.

void * scm_array_handle_uniform_writable_elements (scm_t_array_handle *handle) [C Function]
    Like scm_array_handle_uniform_elements, but the pointer is good reading and writing.

size_t scm_array_handle_uniform_element_size (scm_t_array_handle *handle) [C Function]
    Return the size of one element of the uniform numeric array represented by handle.

const scm_t_uint8 * scm_array_handle_u8_elements (scm_t_array_handle *handle) [C Function]
const scm_t_int8 * scm_array_handle_s8_elements (scm_t_array_handle *handle) [C Function]
const scm_t_uint16 * scm_array_handle_u16_elements (scm_t_array_handle *handle) [C Function]
const scm_t_int16 * scm_array_handle_s16_elements (scm_t_array_handle *handle) [C Function]
const scm_t_uint32 * scm_array_handle_u32_elements (scm_t_array_handle *handle) [C Function]
const scm_t_int32 * scm_array_handle_s32_elements (scm_t_array_handle *handle) [C Function]
const scm_t_uint64 * scm_array_handle_u64_elements (scm_t_array_handle *handle) [C Function]
const scm_t_int64 * scm_array_handle_s64_elements (scm_t_array_handle *handle) [C Function]
const float * scm_array_handle_f32_elements (scm_t_array_handle *handle) [C Function]
Return a pointer to the elements of a uniform numeric array of the indicated kind for reading. This pointer is valid as long as the array remains reserved.

The pointers for c32 and c64 uniform numeric arrays point to pairs of floating point numbers. The even index holds the real part, the odd index the imaginary part of the complex number.

Like scm_array_handle_<kind>_elements, but the pointer is good for reading and writing.

Return a pointer to the words that store the bits of the represented array, which must be a bit array.

Unlike other arrays, bit arrays have an additional offset that must be figured into index calculations. That offset is returned by scm_array_handle_bit_elements_offset.
To find a certain bit you first need to calculate its position as explained above for `scm_array_handle_dims` and then add the offset. This gives the absolute position of the bit, which is always a non-negative integer.

Each word of the bit array storage block contains exactly 32 bits, with the least significant bit in that word having the lowest absolute position number. The next word contains the next 32 bits.

Thus, the following code can be used to access a bit whose position according to `scm_array_handle_dims` is given in `pos`:

```c
SCM bit_array;
scm_t_array_handle handle;
scm_t_uint32 *bits;
ssize_t pos;
size_t abs_pos;
size_t word_pos, mask;

scm_array_get_handle (&bit_array, &handle);
bits = scm_array_handle_bit_elements (&handle);

pos = ...  // Example position calculation
abs_pos = pos + scm_array_handle_bit_elements_offset (&handle);
word_pos = abs_pos / 32;
mask = 1L << (abs_pos % 32);

if (bits[word_pos] & mask)
   /* bit is set. */

scm_array_handle_release (&handle);
```

**6.6.14 VLists**

The (ice-9 vlist) module provides an implementation of the VList data structure designed by Phil Bagwell in 2002. VLists are immutable lists, which can contain any Scheme object. They improve on standard Scheme linked lists in several areas:

- Random access has typically constant-time complexity.
- Computing the length of a VList has time complexity logarithmic in the number of elements.
- VLists use less storage space than standard lists.
- VList elements are stored in contiguous regions, which improves memory locality and leads to more efficient use of hardware caches.

The idea behind VLists is to store vlist elements in increasingly large contiguous blocks (implemented as vectors here). These blocks are linked to one another using a pointer to
the next block and an offset within that block. The size of these blocks form a geometric series with ratio \texttt{block-growth-factor} (2 by default).

The VList structure also serves as the basis for the \texttt{VList-based hash lists} or “vhashes”, an immutable dictionary type (see Section 6.6.21 [VHashes], page 235).

However, the current implementation in \texttt{(ice-9 vlist)} has several noteworthy shortcomings:

- It is not thread-safe. Although operations on vlists are all referentially transparent (i.e., purely functional), adding elements to a vlist with \texttt{vlist-cons} mutates part of its internal structure, which makes it non-thread-safe. This could be fixed, but it would slow down \texttt{vlist-cons}.

- \texttt{vlist-cons} always allocates at least as much memory as \texttt{cons}. Again, Phil Bagwell describes how to fix it, but that would require tuning the garbage collector in a way that may not be generally beneficial.

- \texttt{vlist-cons} is a Scheme procedure compiled to bytecode, and it does not compete with the straightforward C implementation of \texttt{cons}, and with the fact that the VM has a special \texttt{cons} instruction.

We hope to address these in the future.

The programming interface exported by \texttt{(ice-9 vlist)} is defined below. Most of it is the same as SRFI-1 with an added \texttt{vlist-} prefix to function names.

\begin{verbatim}
\[Scheme Procedure\]
\textbf{vlist?} \textit{obj}
\textnormal{Return true if \textit{obj} is a VList.}

\[Scheme Variable\]
\textbf{vlist-null}
\textnormal{The empty VList. Note that it’s possible to create an empty VList not eq? to \texttt{vlist-null}; thus, callers should always use \texttt{vlist-null?} when testing whether a VList is empty.}

\[Scheme Procedure\]
\textbf{vlist-null?} \textit{vlist}
\textnormal{Return true if \textit{vlist} is empty.}

\[Scheme Procedure\]
\textbf{vlist-cons} \textit{item} \textit{vlist}
\textnormal{Return a new vlist with \textit{item} as its head and \textit{vlist} as its tail.}

\[Scheme Procedure\]
\textbf{vlist-head} \textit{vlist}
\textnormal{Return the head of \textit{vlist}.}

\[Scheme Procedure\]
\textbf{vlist-tail} \textit{vlist}
\textnormal{Return the tail of \textit{vlist}.}

\[Scheme Variable\]
\textbf{block-growth-factor}
\textnormal{A fluid that defines the growth factor of VList blocks, 2 by default.}
\end{verbatim}

The functions below provide the usual set of higher-level list operations.

\begin{verbatim}
\[Scheme Procedure\]
\textbf{vlist-fold} \textit{proc} \textit{init} \textit{vlist}
\textnormal{Fold over \textit{vlist}, calling \textit{proc} for each element, as for SRFI-1 \texttt{fold} and \texttt{fold-right} (see Section 7.5.3 [SRFI-1], page 578).}
\end{verbatim}
vlist-ref vlist index
Return the element at index index in vlist. This is typically a constant-time operation.

vlist-length vlist
Return the length of vlist. This is typically logarithmic in the number of elements in vlist.

vlist-reverse vlist
Return a new vlist whose content are those of vlist in reverse order.

vlist-map proc vlist
Map proc over the elements of vlist and return a new vlist.

vlist-for-each proc vlist
Call proc on each element of vlist. The result is unspecified.

vlist-drop vlist count
Return a new vlist that does not contain the count first elements of vlist. This is typically a constant-time operation.

vlist-take vlist count
Return a new vlist that contains only the count first elements of vlist.

vlist-filter pred vlist
Return a new vlist containing all the elements from vlist that satisfy pred.

vlist-delete x vlist [equal?]
Return a new vlist corresponding to vlist without the elements equal? to x.

vlist-unfold p f g seed [tail-gen]
Return a new vlist, as for SRFI-1 unfold and unfold-right (see Section 7.5.3 [SRFI-1], page 578).

vlist-unfold-right p f g seed [tail]
Return a new vlist, as for SRFI-1 unfold and unfold-right (see Section 7.5.3 [SRFI-1], page 578).

vlist-append vlist ...
Append the given vlists and return the resulting vlist.

list->vlist lst
Return a new vlist whose contents correspond to lst.

vlist->list vlist
Return a new list whose contents match those of vlist.

6.6.15 Record Overview
Records, also called structures, are Scheme’s primary mechanism to define new disjoint types. A record type defines a list of fields that instances of the type consist of. This is like C’s struct.

Historically, Guile has offered several different ways to define record types and to create records, offering different features, and making different trade-offs. Over the years, each “standard” has also come with its own new record interface, leading to a maze of record APIs.
At the highest level is SRFI-9, a high-level record interface implemented by most Scheme implementations (see Section 6.6.16 [SRFI-9 Records], page 218). It defines a simple and efficient syntactic abstraction of record types and their associated type predicate, fields, and field accessors. SRFI-9 is suitable for most uses, and this is the recommended way to create record types in Guile. Similar high-level record APIs include SRFI-35 (see Section 7.5.24 [SRFI-35], page 620) and R6RS records (see Section 7.6.2.9 [rnrs records syntactic], page 662).

Then comes Guile’s historical “records” API (see Section 6.6.17 [Records], page 221). Record types defined this way are first-class objects. Introspection facilities are available, allowing users to query the list of fields or the value of a specific field at run-time, without prior knowledge of the type.

Finally, the common denominator of these interfaces is Guile’s structure API (see Section 6.6.18 [Structures], page 222). Guile’s structures are the low-level building block for all other record APIs. Application writers will normally not need to use it.

Records created with these APIs may all be pattern-matched using Guile’s standard pattern matcher (see Section 7.7 [Pattern Matching], page 691).

### 6.6.16 SRFI-9 Records

SRFI-9 standardizes a syntax for defining new record types and creating predicate, constructor, and field getter and setter functions. In Guile this is the recommended option to create new record types (see Section 6.6.15 [Record Overview], page 217). It can be used with:

```scheme
(use-modules (srfi srfi-9))

define-record-type type
  (constructor fieldname ...)    ; [Scheme Syntax]
  predicate
    (fieldname accessor [modifier]) ...
```

Create a new record type, and make various `define`s for using it. This syntax can only occur at the top-level, not nested within some other form.

- `type` is bound to the record type, which is as per the return from the core `make-record-type`. `type` also provides the name for the record, as per `record-type-name`.
- `constructor` is bound to a function to be called as `constructor fieldval ...` to create a new record of this type. The arguments are initial values for the fields, one argument for each field, in the order they appear in the `define-record-type` form.
- The `fieldnames` provide the names for the record fields, as per the core `record-type-fields` etc, and are referred to in the subsequent accessor/modifier forms.
- `predicate` is bound to a function to be called as `(predicate obj)`. It returns `#t` or `#f` according to whether `obj` is a record of this type.
- Each `accessor` is bound to a function to be called `(accessor record)` to retrieve the respective field from a `record`. Similarly each `modifier` is bound to a function to be called `(modifier record val)` to set the respective field in a `record`. 
An example will illustrate typical usage,

\[
\begin{align*}
\text{(define-record-type <employee>}
\text{(make-employee name age salary)}
\text{employee?}
\text{ (name employee-name)}
\text{ (age employee-age set-employee-age!)}
\text{ (salary employee-salary set-employee-salary!))}
\end{align*}
\]

This creates a new employee data type, with name, age and salary fields. Accessor functions are created for each field, but no modifier function for the name (the intention in this example being that it’s established only when an employee object is created). These can all then be used as for example,

\[
\begin{align*}
\text{<employee>} & \Rightarrow #\langle\text{record-type <employee>}\rangle \\
\text{(define fred (make-employee "Fred" 45 20000.00))}
\end{align*}
\]

\[
\begin{align*}
\text{(employee? fred)} & \Rightarrow #t \\
\text{(employee-age fred)} & \Rightarrow 45 \\
\text{(set-employee-salary! fred 25000.00)} & \text{;; pay rise}
\end{align*}
\]

The functions created by \text{define-record-type} are ordinary top-level \text{defines}. They can be redefined or \text{set!} as desired, exported from a module, etc.

**Non-toplevel Record Definitions**

The SRFI-9 specification explicitly disallows record definitions in a non-toplevel context, such as inside \text{lambda} body or inside a \text{let} block. However, Guile’s implementation does not enforce that restriction.

**Custom Printers**

You may use \text{set-record-type-printer!} to customize the default printing behavior of records. This is a Guile extension and is not part of SRFI-9. It is located in the \text{(srfi srfi-9 gnu)} module.

\text{set-record-type-printer! type proc} \quad \text{[Scheme Syntax]}

Where \text{type} corresponds to the first argument of \text{define-record-type}, and \text{proc} is a procedure accepting two arguments, the record to print, and an output port.

This example prints the employee’s name in brackets, for instance [Fred].

\[
\begin{align*}
\text{(set-record-type-printer! <employee>}
\text{(lambda (record port)
 (write-char #\[ port)
 (display (employee-name record) port)
 (write-char #\] port)))}
\end{align*}
\]

**Functional “Setters”**

When writing code in a functional style, it is desirable to never alter the contents of records. For such code, a simple way to return new record instances based on existing ones is highly desirable.
The \texttt{(srfi srfi-9 gnu)} module extends SRFI-9 with facilities to return new record instances based on existing ones, only with one or more field values changed—\texttt{functional setters}. First, the \texttt{define-immutable-record-type} works like \texttt{define-record-type}, except that fields are immutable and setters are defined as functional setters.

\begin{verbatim}
define-immutable-record-type type
   (constructor fieldname ...) predicate
   (fieldname accessor [modifier]) ...

Define \textit{type} as a new record type, like \texttt{define-record-type}. However, the record type is made \textit{immutable} (records may not be mutated, even with \texttt{struct-set!}), and any \textit{modifier} is defined to be a functional setter—a procedure that returns a new record instance with the specified field changed, and leaves the original unchanged (see example below.)

In addition, the generic \texttt{set-field} and \texttt{set-fields} macros may be applied to any SRFI-9 record.

\begin{verbatim}
set-field record (field sub-fields ...) value
Return a new record of \textit{record}'s type whose fields are equal to the corresponding fields of \textit{record} except for the one specified by \textit{field}. \textit{field} must be the name of the getter corresponding to the field of \textit{record} being “set”. Subsequent \textit{sub-fields} must be record getters designating sub-fields within that field value to be set (see example below.)

set-fields record ((field sub-fields ...) value) ...
Like \texttt{set-field}, but can be used to set more than one field at a time. This expands to code that is more efficient than a series of single \texttt{set-field} calls.

To illustrate the use of functional setters, let's assume these two record type definitions:

\begin{verbatim}
(define-record-type <address>
   (address street city country)
   address?
   (street address-street)
   (city address-city)
   (country address-country))

(define-immutable-record-type <person>
   (person age email address)
   person?
   (age person-age set-person-age)
   (email person-email set-person-email)
   (address person-address set-person-address))
\end{verbatim}

First, note that the \texttt{<person>} record type definition introduces named functional setters. These may be used like this:

\begin{verbatim}
(define fsf-address
   (address "Franklin Street" "Boston" "USA"))
\end{verbatim}
\end{verbatim}
(define rms
  (person 30 "rms@gnu.org" fsf-address))

(and (equal? (set-person-age rms 60)
             (person 60 "rms@gnu.org" fsf-address))
      (= (person-age rms) 30))
⇒ #t

Here, the original <person> record, to which rms is bound, is left unchanged.

Now, suppose we want to change both the street and age of rms. This can be achieved using set-fields:

(set-fields rms
  ((person-age) 60)
  ((person-address address-street) "Temple Place")
⇒ #<<person> age: 60 email: "rms@gnu.org"
                           address: #<<address> street: "Temple Place" city: "Boston" country: "USA">>

Notice how the above changed two fields of rms, including the street field of its address field, in a concise way. Also note that set-fields works equally well for types defined with just define-record-type.

6.6.17 Records

A record type is a first class object representing a user-defined data type. A record is an instance of a record type.

Note that in many ways, this interface is too low-level for every-day use. Most uses of records are better served by SRFI-9 records. See Section 6.6.16 [SRFI-9 Records], page 218.

record? obj
  Return #t if obj is a record of any type and #f otherwise.
  Note that record? may be true of any Scheme value; there is no promise that records are disjoint with other Scheme types.

make-record-type type-name field-names [print]
  Create and return a new record-type descriptor.
  type-name is a string naming the type. Currently it’s only used in the printed representation of records, and in diagnostics. field-names is a list of symbols naming the fields of a record of the type. Duplicates are not allowed among these symbols.

  (make-record-type "employee" '(name age salary))

  The optional print argument is a function used by display, write, etc, for printing a record of the new type. It’s called as (print record port) and should look at record and write to port.

record-constructor rtd [field-names]
  Return a procedure for constructing new members of the type represented by rtd. The returned procedure accepts exactly as many arguments as there are symbols in the given list, field-names; these are used, in order, as the initial values of those fields in a new record, which is returned by the constructor procedure. The values of any fields not named in that list are unspecified. The field-names argument defaults to the list
of field names in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}; if the \texttt{field-names} argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

\begin{description}
\item[record-predicate \texttt{rtd}] [Scheme Procedure]
Return a procedure for testing membership in the type represented by \texttt{rtd}. The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type; it returns a false value otherwise.

\item[record-accessor \texttt{rtd field-name}] [Scheme Procedure]
Return a procedure for reading the value of a particular field of a member of the type represented by \texttt{rtd}. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol \texttt{field-name} in that record. The symbol \texttt{field-name} must be a member of the list of field-names in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}.

\item[record-modifier \texttt{rtd field-name}] [Scheme Procedure]
Return a procedure for writing the value of a particular field of a member of the type represented by \texttt{rtd}. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol \texttt{field-name} in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol \texttt{field-name} must be a member of the list of field-names in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}.

\item[record-type-descriptor \texttt{record}] [Scheme Procedure]
Return a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to \texttt{record-predicate}, the resulting predicate would return a true value when passed the given record. Note that it is not necessarily the case that the returned descriptor is the one that was passed to \texttt{record-constructor} in the call that created the constructor procedure that created the given record.

\item[record-type-name \texttt{rtd}] [Scheme Procedure]
Return the type-name associated with the type represented by \texttt{rtd}. The returned value is \texttt{eqv?} to the \texttt{type-name} argument given in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}.

\item[record-type-fields \texttt{rtd}] [Scheme Procedure]
Return a list of the symbols naming the fields in members of the type represented by \texttt{rtd}. The returned value is \texttt{equal?} to the field-names argument given in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}.
\end{description}

\section*{6.6.18 Structures}

A \textit{structure} is a first class data type which holds Scheme values or C words in fields numbered 0 upwards. A \textit{vtable} is a structure that represents a structure type, giving field types and permissions, and an optional print function for \texttt{write} etc.
Structures are lower level than records (see Section 6.6.17 [Records], page 221). Usually, when you need to represent structured data, you just want to use records. But sometimes you need to implement new kinds of structured data abstractions, and for that purpose structures are useful. Indeed, records in Guile are implemented with structures.

6.6.18.1 Vtables

A vtable is a structure type, specifying its layout, and other information. A vtable is actually itself a structure, but there’s no need to worry about that initially (see Section 6.6.18.3 [Vtable Contents], page 225.)

**make-vtable fields [print]**

Create a new vtable.

fields is a string describing the fields in the structures to be created. Each field is represented by two characters, a type letter and a permissions letter, for example "pw". The types are as follows.

- **p** – a Scheme value. “p” stands for “protected” meaning it’s protected against garbage collection.
- **u** – an arbitrary word of data (an scm_t_bits). At the Scheme level it’s read and written as an unsigned integer. “u” stands for “unboxed”, as it’s stored as a raw value without additional type annotations.

The second letter for each field is a permission code,

- **w** – writable, the field can be read and written.
- **r** – read-only, the field can be read but not written.

Here are some examples.

```
(make-vtable "pw") ;; one writable field
(make-vtable "prpw") ;; one read-only and one writable
(make-vtable "pwuwuw") ;; one scheme and two unboxed
```

The optional print argument is a function called by display and write (etc) to give a printed representation of a structure created from this vtable. It’s called (print struct port) and should look at struct and write to port. The default print merely gives a form like ‘#<struct ADDR:ADDR>’ with a pair of machine addresses.

The following print function for example shows the two fields of its structure.

```
(make-vtable "prpw"
  (lambda (struct port)
    (format port "#<~a and ~a>
      (struct-ref struct 0)
      (struct-ref struct 1))))
```

6.6.18.2 Structure Basics

This section describes the basic procedures for working with structures. make-struct/no-tail creates a structure, and struct-ref and struct-set! access its fields.
**make-struct/no-tail vtable init . . .**

Create a new structure, with layout per the given vtable (see Section 6.6.18.1 [Vtables], page 223).

The optional init . . . arguments are initial values for the fields of the structure. This is the only way to put values in read-only fields. If there are fewer init arguments than fields then the defaults are #f for a Scheme field (type p) or 0 for an unboxed field (type u).

The name is a bit strange, we admit. The reason for it is that Guile used to have a make-struct that took an additional argument; while we deprecate that old interface, make-struct/no-tail is the new name for this functionality.

For example,

```
(define v (make-vtable "prpwpw"))
(define s (make-struct/no-tail v 123 "abc" 456))
(struct-ref s 0) ⇒ 123
(struct-ref s 1) ⇒ "abc"
```

**SCM scm_make_struct (SCM vtable, SCM tail-size, SCM init-list)**

[C Function]

There are a few ways to make structures from C. scm_make_struct takes a list, scm_c_make_struct takes variable arguments terminated with SCM_UNDEFINED, and scm_c_make_structv takes a packed array.

For all of these, tail_size should be zero (as a SCM value).

**struct? obj**

[Scheme Procedure]

scm_struct_p (obj)

[C Function]

Return #t if obj is a structure, or #f if not.

**struct-ref struct n**

[Scheme Procedure]

scm_struct_ref (struct, n)

[C Function]

Return the contents of field number n in struct. The first field is number 0.

An error is thrown if n is out of range.

**struct-set! struct n value**

[Scheme Procedure]

scm_struct_set_x (struct, n, value)

[C Function]

Set field number n in struct to value. The first field is number 0.

An error is thrown if n is out of range, or if the field cannot be written because it’s read-only.

Unboxed fields (those with type u) need to be accessed with special procedures.

**struct-ref/unboxed struct n**

[Scheme Procedure]

**struct-set!/unboxed struct n value**

[Scheme Procedure]

**scm_struct_ref_unboxed (struct, n)**

[C Function]
**scm_struct_set_x_unboxed** \(\text{(struct, n, value)}\) \[C\ Function\]

Like **struct-ref** and **struct-set!**, except that these may only be used on unboxed fields. **struct-ref/unboxed** will always return a positive integer. Likewise, **struct-set!/unboxed** takes an unsigned integer as the value argument, and will signal an error otherwise.

**struct-vtable** **struct** \[Scheme Procedure\]
**scm_struct_vtable** **(struct)** \[C Function\]

Return the vtable that describes **struct**.

The vtable is effectively the type of the structure. See Section 6.6.18.3 [Vtable Contents], page 225, for more on vtables.

### 6.6.18.3 Vtable Contents

A vtable is itself a structure. It has a specific set of fields describing various aspects of its instances: the structures created from a vtable. Some of the fields are internal to Guile, some of them are part of the public interface, and there may be additional fields added on by the user.

Every vtable has a field for the layout of their instances, a field for the procedure used to print its instances, and a field for the name of the vtable itself. Access to the layout and printer is exposed directly via field indexes. Access to the vtable name is exposed via accessor procedures.

**vtable-index-layout** \[Scheme Variable\]
**scm_vtable_index_layout** \[C Macro\]

The field number of the layout specification in a vtable. The layout specification is a symbol like `pwpw` formed from the fields string passed to **make-vtable**, or created by **make-struct-layout** (see Section 6.6.18.4 [Meta-Vtables], page 226).

\[
\text{(define v (make-vtable "pwpw" 0))}
\]

\[
\text{(struct-ref v vtable-index-layout) \Rightarrow pwpw}
\]

This field is read-only, since the layout of structures using a vtable cannot be changed.

**vtable-index-printer** \[Scheme Variable\]
**scm_vtable_index_printer** \[C Macro\]

The field number of the printer function. This field contains `#f` if the default print function should be used.

\[
\text{(define (my-print-func struct port)}
\text{ ...)}
\]

\[
\text{(define v (make-vtable "pwpw" my-print-func))}
\]

\[
\text{(struct-ref v vtable-index-printer) \Rightarrow my-print-func}
\]

This field is writable, allowing the print function to be changed dynamically.

**struct-vtable-name** **vtable** \[Scheme Procedure\]
**set-struct-vtable-name!** **vtable name** \[Scheme Procedure\]
**scm_struct_vtable_name** **(vtable)** \[C Function\]
**scm_set_struct_vtable_name_x** **(vtable, name)** \[C Function\]

Get or set the name of **vtable**. **name** is a symbol and is used in the default print function when printing structures created from **vtable**.

\[
\text{(define v (make-vtable "pw"))}
\]
(set-struct-vtable-name! v 'my-name)

(define s (make-struct v 0))
(display s) → #<my-name b7ab3ae0:b7ab3730>

6.6.18.4 Meta-Vtables

As a structure, a vtable also has a vtable, which is also a structure. Structures, their vtables, the vtables of the vtables, and so on form a tree of structures. Making a new structure adds a leaf to the tree, and if that structure is a vtable, it may be used to create other leaves.

If you traverse up the tree of vtables, via calling struct-vtable, eventually you reach a root which is the vtable of itself:

```
scheme@(guile-user)> (current-module)
$1 = #<directory (guile-user) 221b090>
scheme@(guile-user)> (struct-vtable $1)
$2 = #<record-type module>
scheme@(guile-user)> (struct-vtable $2)
$3 = #<standard-vtable 12c30a0>
scheme@(guile-user)> (struct-vtable $3)
$4 = #<standard-vtable 12c3fa0>
scheme@(guile-user)> (struct-vtable $4)
$5 = #<standard-vtable 12c3fa0>
scheme@(guile-user)> <standard-vtable>
$6 = #<standard-vtable 12c3fa0>
```

In this example, we can say that $1$ is an instance of $2$, $2$ is an instance of $3$, $3$ is an instance of $4$, and $4$, strangely enough, is an instance of itself. The value bound to $4$ in this console session also bound to $<\text{standard-vtable}>$ in the default environment.

$<\text{standard-vtable}>$

A meta-vtable, useful for making new vtables.

All of these values are structures. All but $1$ are vtables. As $2$ is an instance of $3$, and $3$ is a vtable, we can say that $3$ is a meta-vtable: a vtable that can create vtables.

With this definition, we can specify more precisely what a vtable is: a vtable is a structure made from a meta-vtable. Making a structure from a meta-vtable runs some special checks to ensure that the first field of the structure is a valid layout. Additionally, if these checks see that the layout of the child vtable contains all the required fields of a vtable, in the correct order, then the child vtable will also be a meta-table, inheriting a magical bit from the parent.

```
struct-vtable? obj
```

[Scheme Procedure]

```
scm_struct_vtable_p (obj)
```

[C Function]

Return #t if obj is a vtable structure: an instance of a meta-vtable.

$<\text{standard-vtable}>$ is a root of the vtable tree. (Normally there is only one root in a given Guile process, but due to some legacy interfaces there may be more than one.)

The set of required fields of a vtable is the set of fields in the $<\text{standard-vtable}>$, and is bound to $\text{standard-vtable-fields}$ in the default environment. It is possible to create
a meta-vtable that with additional fields in its layout, which can be used to create vtables with additional data:

```scheme
scheme@(guile-user)> (struct-ref $3 vtable-index-layout)
$6 = pruhsruhpwpuhuhprprpw
scheme@(guile-user)> (struct-ref $4 vtable-index-layout)
$7 = pruhsruhpwpuhuh
scheme@(guile-user)> standard-vtable-fields
$8 = "pruhsruhpwpuhuh"
scheme@(guile-user)> (struct-ref $2 vtable-offset-user)
$9 = module
```

In this continuation of our earlier example, $2$ is a vtable that has extra fields, because its vtable, $3$, was made from a meta-vtable with an extended layout. `vtable-offset-user` is a convenient definition that indicates the number of fields in `standard-vtable-fields`.

- **standard-vtable-fields**  
  A string containing the ordered set of fields that a vtable must have.

- **vtable-offset-user**  
  The first index in a vtable that is available for a user.

- **make-struct-layout fields**  
  [Scheme Procedure]

- **scm_make_struct_layout**  
  [C Function]
  
  Return a structure layout symbol, from a `fields` string. `fields` is as described under `make-vtable` (see Section 6.6.18.1 [Vtables], page 223). An invalid `fields` string is an error.

With these definitions, one can define `make-vtable` in this way:

```scheme
(define* (make-vtable fields #:optional printer)
  (make-struct/no-tail <standard-vtable>
   (make-struct-layout fields)
   printer))
```

### 6.6.18.5 Vtable Example

Let us bring these points together with an example. Consider a simple object system with single inheritance. Objects will be normal structures, and classes will be vtables with three extra class fields: the name of the class, the parent class, and the list of fields.

So, first we need a meta-vtable that allocates instances with these extra class fields.

```scheme
(define <class>
  (make-vtable
   (string-append standard-vtable-fields "pwpwpw")
   (lambda (x port)
     (format port "<<class> ~a>" (class-name x))))))

(define (class? x)
  (and (struct? x)
       (eq? (struct-vtable x) <class>)))
```
To make a structure with a specific meta-vtable, we will use `make-struct/no-tail`, passing it the computed instance layout and printer, as with `make-vtable`, and additionally the extra three class fields.

```
(define (make-class name parent fields)
  (let* ((fields (compute-fields parent fields))
          (layout (compute-layout fields)))
    (make-struct/no-tail <class>
      layout
      (lambda (x port)
        (print-instance x port))
      name
      parent
      fields)))
```

Instances will store their associated data in slots in the structure: as many slots as there are fields. The `compute-layout` procedure below can compute a layout, and `field-index` returns the slot corresponding to a field.

```
(define-syntax-rule (define-accessor name n)
  (define (name obj)
    (struct-ref obj n)))
```

;; Accessors for classes
(define-accessor class-name (+ vtable-offset-user 0))
(define-accessor class-parent (+ vtable-offset-user 1))
(define-accessor class-fields (+ vtable-offset-user 2))

(define (compute-fields parent fields)
  (if parent
      (append (class-fields parent) fields)
      fields))

(define (compute-layout fields)
  (make-struct-layout
   (string-concatenate (make-list (length fields) "pw")))))

(define (field-index class field)
  (list-index (class-fields class) field))

(define (print-instance x port)
  (format port "<"a" (class-name (struct-vtable x)))
  (for-each (lambda (idx)
              (format port " ~a: ~a" field (struct-ref x idx))
              (class-fields (struct-vtable x))
              (iota (length (class-fields (struct-vtable x))))
              (format port "">"))
    (field-index (struct-vtable x)))
  (format port ">"))
```

So, at this point we can actually make a few classes:

```
(define-syntax-rule (define-class name parent field ...)
```

(define name (make-class 'name parent '(field ...))))

(define-class <surface> #f
    width height)

(define-class <window> <surface>
    x y)
And finally, make an instance:
(make-struct/no-tail <window> 400 300 10 20)
⇒ <window> width: 400 height: 300 x: 10 y: 20>

And that’s that. Note that there are many possible optimizations and feature enhancements that can be made to this object system, and the included GOOPS system does make most of them. For more simple use cases, the records facility is usually sufficient. But sometimes you need to make new kinds of data abstractions, and for that purpose, structs are here.

6.6.19 Dictionary Types

A dictionary object is a data structure used to index information in a user-defined way. In standard Scheme, the main aggregate data types are lists and vectors. Lists are not really indexed at all, and vectors are indexed only by number (e.g. (vector-ref foo 5)). Often you will find it useful to index your data on some other type; for example, in a library catalog you might want to look up a book by the name of its author. Dictionaries are used to help you organize information in such a way.

An association list (or alist for short) is a list of key-value pairs. Each pair represents a single quantity or object; the car of the pair is a key which is used to identify the object, and the cdr is the object’s value.

A hash table also permits you to index objects with arbitrary keys, but in a way that makes looking up any one object extremely fast. A well-designed hash system makes hash table lookups almost as fast as conventional array or vector references.

Alists are popular among Lisp programmers because they use only the language’s primitive operations (lists, car, cdr and the equality primitives). No changes to the language core are necessary. Therefore, with Scheme’s built-in list manipulation facilities, it is very convenient to handle data stored in an association list. Also, alists are highly portable and can be easily implemented on even the most minimal Lisp systems.

However, alists are inefficient, especially for storing large quantities of data. Because we want Guile to be useful for large software systems as well as small ones, Guile provides a rich set of tools for using either association lists or hash tables.

6.6.20 Association Lists

An association list is a conventional data structure that is often used to implement simple key-value databases. It consists of a list of entries in which each entry is a pair. The key of each entry is the car of the pair and the value of each entry is the cdr.

ASSOCIATION LIST ::= '( (KEY1 . VALUE1)
                       (KEY2 . VALUE2)
                       (KEY3 . VALUE3)
Association lists are also known, for short, as *alists*.

The structure of an association list is just one example of the infinite number of possible structures that can be built using pairs and lists. As such, the keys and values in an association list can be manipulated using the general list structure procedures `cons`, `car`, `cdr`, `set-car!`, `set-cdr!` and so on. However, because association lists are so useful, Guile also provides specific procedures for manipulating them.

### 6.6.20.1 Alist Key Equality

All of Guile’s dedicated association list procedures, apart from `acons`, come in three flavours, depending on the level of equality that is required to decide whether an existing key in the association list is the same as the key that the procedure call uses to identify the required entry.

- Procedures with `assq` in their name use `eq?` to determine key equality.
- Procedures with `assv` in their name use `eqv?` to determine key equality.
- Procedures with `assoc` in their name use `equal?` to determine key equality.

`acons` is an exception because it is used to build association lists which do not require their entries’ keys to be unique.

### 6.6.20.2 Adding or Setting Alist Entries

`acons` adds a new entry to an association list and returns the combined association list. The combined alist is formed by consing the new entry onto the head of the alist specified in the `acons` procedure call. So the specified alist is not modified, but its contents become shared with the tail of the combined alist that `acons` returns.

In the most common usage of `acons`, a variable holding the original association list is updated with the combined alist:

```scheme
(set! address-list (acons name address address-list))
```

In such cases, it doesn’t matter that the old and new values of `address-list` share some of their contents, since the old value is usually no longer independently accessible.

Note that `acons` adds the specified new entry regardless of whether the alist may already contain entries with keys that are, in some sense, the same as that of the new entry. Thus `acons` is ideal for building alists where there is no concept of key uniqueness.

```scheme
(set! task-list (acons 3 "pay gas bill" '()))
```

```scheme
⇒ ((3 . "pay gas bill"))
```

```scheme
(set! task-list (acons 3 "tidy bedroom" task-list))
```

```scheme
⇒ ((3 . "tidy bedroom") (3 . "pay gas bill"))
```

`assq-set!`, `assv-set!` and `assoc-set!` are used to add or replace an entry in an association list where there is a concept of key uniqueness. If the specified association list

already contains an entry whose key is the same as that specified in the procedure call, the existing entry is replaced by the new one. Otherwise, the new entry is consed onto the head of the old association list to create the combined alist. In all cases, these procedures return the combined alist.

**assq-set!** and friends *may* destructively modify the structure of the old association list in such a way that an existing variable is correctly updated without having to **set!** it to the value returned:

```
address-list
⇒
(("mary" . "34 Elm Road") ("james" . "16 Bow Street"))

(assoc-set! address-list "james" "1a London Road")
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

Or they may not:

```
(assoc-set! address-list "bob" "11 Newington Avenue")
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

The only safe way to update an association list variable when adding or replacing an entry like this is to **set!** the variable to the returned value:

```
(set! address-list
 (assoc-set! address-list "bob" "11 Newington Avenue"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))
```

Because of this slight inconvenience, you may find it more convenient to use hash tables to store dictionary data. If your application will not be modifying the contents of an alist very often, this may not make much difference to you.

If you need to keep the old value of an association list in a form independent from the list that results from modification by **acons**, **assq-set!**, **assv-set!** or **assoc-set!**, use **list-copy** to copy the old association list before modifying it.
acons key value alist

scm_acons (key, value, alist)

Add a new key-value pair to alist. A new pair is created whose car is key and whose cdr is value, and the pair is consed onto alist, and the new list is returned. This function is not destructive; alist is not modified.

assq-set! alist key val
assv-set! alist key value
assoc-set! alist key value

Reassociate key in alist with value: find any existing alist entry for key and associate it with the new value. If alist does not contain an entry for key, add a new one. Return the (possibly new) alist.

These functions do not attempt to verify the structure of alist, and so may cause unusual results if passed an object that is not an association list.

6.6.20.3 Retrieving Alist Entries

assq, assv and assoc find the entry in an alist for a given key, and return the (key . value) pair. assq-ref, assv-ref and assoc-ref do a similar lookup, but return just the value.

assq key alist
assv key alist
assoc key alist

Return the first entry in alist with the given key. The return is the pair (KEY . VALUE) from alist. If there’s no matching entry the return is #f.

assq compares keys with eq?, assv uses eqv? and assoc uses equal?. See also SRFI-1 which has an extended assoc (Section 7.5.3.9 [SRFI-1 Association Lists], page 588).

assq-ref alist key
assv-ref alist key
assoc-ref alist key

Return the value from the first entry in alist with the given key, or #f if there’s no such entry.

assq-ref compares keys with eq?, assv-ref uses eqv? and assoc-ref uses equal?. Notice these functions have the key argument last, like other -ref functions, but this is opposite to what assq etc above use.

When the return is #f it can be either key not found, or an entry which happens to have value #f in the cdr. Use assq etc above if you need to differentiate these cases.
6.6.20.4 Removing Alist Entries

To remove the element from an association list whose key matches a specified key, use assq-remove!, assv-remove! or assoc-remove! (depending, as usual, on the level of equality required between the key that you specify and the keys in the association list).

As with assq-set! and friends, the specified alist may or may not be modified destructively, and the only safe way to update a variable containing the alist is to set! it to the value that assq-remove! and friends return.

address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("james" . "1a London Road"))

Note that, when assq/v/oc-remove! is used to modify an association list that has been constructed only using the corresponding assq/v/oc-set!, there can be at most one matching entry in the alist, so the question of multiple entries being removed in one go does not arise. If assq/v/oc-remove! is applied to an association list that has been constructed using acons, or an assq/v/oc-set! with a different level of equality, or any mixture of these, it removes only the first matching entry from the alist, even if the alist might contain further matching entries. For example:

(define address-list '())
(set! address-list (assq-set! address-list "mary" "11 Elm Street"))
(set! address-list (assq-set! address-list "mary" "57 Pine Drive"))
address-list
⇒
(("mary" . "57 Pine Drive") ("mary" . "11 Elm Street"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("mary" . "11 Elm Street"))

In this example, the two instances of the string "mary" are not the same when compared using eq?, so the two assq-set! calls add two distinct entries to address-list. When compared using equal?, both "mary"s in address-list are the same as the "mary" in the assoc-remove! call, but assoc-remove! stops after removing the first matching entry that it finds, and so one of the "mary" entries is left in place.

assq-remove! alist key [Scheme Procedure]
assv-remove! alist key [Scheme Procedure]
assoc-remove! alist key [Scheme Procedure]
scm_assq_remove_x (alist, key) [C Function]
scm_assv_remove_x (alist, key) [C Function]
scm_assoc_remove_x (alist, key)  
Delete the first entry in alist associated with key, and return the resulting alist.

6.6.20.5 Sloppy Alist Functions

sloppy-assq, sloppy-assv and sloppy-assoc behave like the corresponding non-sloppy-procedures, except that they return #f when the specified association list is not well-formed, where the non-sloppy-procedures would signal an error.

Specifically, there are two conditions for which the non-sloppy-procedures signal an error, which the sloppy-procedures handle instead by returning #f. Firstly, if the specified alist as a whole is not a proper list:

```scheme
(assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
ERROR: In procedure assoc in expression (assoc "mary" (quote #)):
ERROR: Wrong type argument in position 2 (expecting association list): ((1 . 2) ("key" . "door") . "open sesame")
```

```c
(scm_sloppy_assq "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
#f
```

Secondly, if one of the entries in the specified alist is not a pair:

```scheme
(assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
ERROR: In procedure assoc in expression (assoc 2 (quote #)):
ERROR: Wrong type argument in position 2 (expecting association list): ((1 . 1) 2 (3 . 9))
```

```c
(scm_sloppy_assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
#f
```

Unless you are explicitly working with badly formed association lists, it is much safer to use the non-sloppy-procedures, because they help to highlight coding and data errors that the sloppy-procedures would silently cover up.

sloppy-assq key alist  
Behaves like assq but does not do any error checking. Recommended only for use in Guile internals.

sloppy-assv key alist  
Behaves like assv but does not do any error checking. Recommended only for use in Guile internals.

sloppy-assoc key alist  
Behaves like assoc but does not do any error checking. Recommended only for use in Guile internals.
6.6.20.6 Alist Example

Here is a longer example of how alists may be used in practice.

```
(define capitals '((("New York" . "Albany")
                     ("Oregon" . "Salem")
                     ("Florida" . "Miami")))

;; What’s the capital of Oregon?
(assoc "Oregon" capitals) ⇒ ("Oregon" . "Salem")
(assoc-ref capitals "Oregon") ⇒ "Salem"

;; We left out South Dakota.
(set! capitals
    (assoc-set! capitals "South Dakota" "Pierre"))
capitals
⇒ ((("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Miami"))

;; And we got Florida wrong.
(set! capitals
    (assoc-set! capitals "Florida" "Tallahassee"))
capitals
⇒ ((("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Tallahassee"))

;; After Oregon secedes, we can remove it.
(set! capitals
    (assoc-remove! capitals "Oregon"))
capitals
⇒ ((("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Florida" . "Tallahassee"))
```

6.6.21 VList-Based Hash Lists or “VHashes”

The (ice-9 vlist) module provides an implementation of VList-based hash lists (see Section 6.6.14 [VLists], page 215). VList-based hash lists, or vhashes, are an immutable dictionary type similar to association lists that maps keys to values. However, unlike association lists, accessing a value given its key is typically a constant-time operation.

The VHash programming interface of (ice-9 vlist) is mostly the same as that of association lists found in SRFI-1, with procedure names prefixed by vhash- instead of alist- (see Section 7.5.3.9 [SRFI-1 Association Lists], page 588).

In addition, vhashes can be manipulated using VList operations:

```
(vlist-head (vhash-consq 'a 1 vlist-null))
```
(define vh1 (vhash-consq 'b 2 (vhash-consq 'a 1 vlist-null)))
(define vh2 (vhash-consq 'c 3 (vlist-tail vh1)))

(vhash-assq 'a vh2)
⇒ (a . 1)
(vhash-assq 'b vh2)
⇒ #f
(vhash-assq 'c vh2)
⇒ (c . 3)
(vlist->list vh2)
⇒ ((c . 3) (a . 1))

However, keep in mind that procedures that construct new VLists (vlist-map, vlist-filter, etc.) return raw VLists, not vhashes:

(define vh (alist->vhash '(((a . 1) (b . 2) (c . 3)) hashq))
(vhash-assq 'a vh)
⇒ (a . 1)

(define vl
  ;; This will create a raw vlist.
  (vlist-filter (lambda (key+value) (odd? (cdr key+value))) vh))
(vhash-assq 'a vl)
⇒ ERROR: Wrong type argument in position 2

(vlist->list vl)
⇒ ((a . 1) (c . 3))

vhash? obj
   [Scheme Procedure]

Return true if obj is a vhash.

vhash-cons key value vhash [hash-proc]
   [Scheme Procedure]
vhash-consq key value vhash
   [Scheme Procedure]
vhash-consv key value vhash
   [Scheme Procedure]

Return a new hash list based on vhash where key is associated with value, using hash-proc to compute the hash of key. vhash must be either vlist-null or a vhash returned by a previous call to vhash-cons. hash-proc defaults to hash (see Section 6.6.22.2 [Hash Table Reference], page 239). With vhash-consq, the hashq hash function is used; with vhash-consv the hashv hash function is used.

All vhash-cons calls made to construct a vhash should use the same hash-proc. Failing to do that, the result is undefined.

vhash-assoc key vhash [equal? [hash-proc]]
   [Scheme Procedure]
vhash-assq key vhash
   [Scheme Procedure]
vhash-assv key vhash
   [Scheme Procedure]

Return the first key/value pair from vhash whose key is equal to key according to the equal? equality predicate (which defaults to equal?), and using hash-proc (which
Chapter 6: API Reference

Chapter 6: API Reference

defaults to \texttt{hash}) to compute the hash of \texttt{key}. The second form uses \texttt{eq?} as the equality predicate and \texttt{hashq} as the hash function; the last form uses \texttt{eqv?} and \texttt{hashv}.

Note that it is important to consistently use the same hash function for \texttt{hash-proc} as was passed to \texttt{vhash-cons}. Failing to do that, the result is unpredictable.

\begin{verbatim}
\texttt{vhash-delete} key vhash \texttt{[equal? \ [hash-proc]]}
\texttt{vhash-delq} key vhash
\texttt{vhash-delv} key vhash
\end{verbatim}

Remove all associations from \texttt{vhash} with \texttt{key}, comparing keys with \texttt{equal?} (which defaults to \texttt{equal?}), and computing the hash of \texttt{key} using \texttt{hash-proc} (which defaults to \texttt{hash}). The second form uses \texttt{eq?} as the equality predicate and \texttt{hashq} as the hash function; the last one uses \texttt{eqv?} and \texttt{hashv}.

Again the choice of \texttt{hash-proc} must be consistent with previous calls to \texttt{vhash-cons}.

\begin{verbatim}
\texttt{vhash-fold} proc init vhash
\texttt{vhash-fold-right} proc init vhash
\texttt{vhash-fold*} proc init key vhash \texttt{[equal? \ [hash]]}
\texttt{vhash-foldq*} proc init key vhash
\texttt{vhash-foldv*} proc init key vhash
\end{verbatim}

Fold over the key/value elements of \texttt{vhash} in the given direction, with each call to \texttt{proc} having the form \texttt{(proc key value result)}, where \texttt{result} is the result of the previous call to \texttt{proc} and \texttt{init} the value of \texttt{result} for the first call to \texttt{proc}.

Keys in \texttt{vhash} are hashed using \texttt{hash} are compared using \texttt{equal?}. The second form uses \texttt{eq?} as the equality predicate and \texttt{hashq} as the hash function; the third one uses \texttt{eqv?} and \texttt{hashv}.

Example:

\begin{verbatim}
(define vh
  (alist->vhash '((a . 1) (a . 2) (z . 0) (a . 3))))

(vhash-fold* cons '() 'a vh)
⇒ (3 2 1)

(vhash-fold* cons '() 'z vh)
⇒ (0)
\end{verbatim}

\begin{verbatim}
\texttt{alist->vhash} alist \texttt{[hash-proc]}
\end{verbatim}

Return the \texttt{vhash} corresponding to \texttt{alist}, an association list, using \texttt{hash-proc} to compute key hashes. When omitted, \texttt{hash-proc} defaults to \texttt{hash}.

6.6.22 Hash Tables

Hash tables are dictionaries which offer similar functionality as association lists: They provide a mapping from keys to values. The difference is that association lists need time linear in the size of elements when searching for entries, whereas hash tables can normally
search in constant time. The drawback is that hash tables require a little bit more memory, and that you can not use the normal list procedures (see Section 6.6.9 [Lists], page 181) for working with them.

6.6.22.1 Hash Table Examples

For demonstration purposes, this section gives a few usage examples of some hash table procedures, together with some explanation what they do.

First we start by creating a new hash table with 31 slots, and populate it with two key/value pairs.

```
(define h (make-hash-table 31))

;; This is an opaque object
h
⇒
#<hash-table 0/31>

;; Inserting into a hash table can be done with hashq-set!
(hashq-set! h 'foo "bar")
⇒
"bar"

(hashq-set! h 'braz "zonk")
⇒
"zonk"

;; Or with hash-create-handle!
(hashq-create-handle! h 'frob #f)
⇒
(frob . #f)
```

You can get the value for a given key with the procedure hashq-ref, but the problem with this procedure is that you cannot reliably determine whether a key does exists in the table. The reason is that the procedure returns #f if the key is not in the table, but it will return the same value if the key is in the table and just happens to have the value #f, as you can see in the following examples.

```
(hashq-ref h 'foo)
⇒
"bar"

(hashq-ref h 'frob)
⇒
#f

(hashq-ref h 'not-there)
⇒
#f
```
It is often better to use the procedure `hashq-get-handle`, which makes a distinction between the two cases. Just like `assq`, this procedure returns a key/value-pair on success, and `#f` if the key is not found.

```lisp
(hashq-get-handle h 'foo)
⇒
  (foo . "bar")
```

```lisp
(hashq-get-handle h 'not-there)
⇒
  #f
```

Interesting results can be computed by using `hash-fold` to work through each element. This example will count the total number of elements:

```lisp
(hash-fold (lambda (key value seed) (+ 1 seed)) 0 h)
⇒
  3
```

The same thing can be done with the procedure `hash-count`, which can also count the number of elements matching a particular predicate. For example, count the number of elements with string values:

```lisp
(hash-count (lambda (key value) (string? value)) h)
⇒
  2
```

Counting all the elements is a simple task using `const`:

```lisp
(hash-count (const #t) h)
⇒
  3
```

### 6.6.22.2 Hash Table Reference

Like the association list functions, the hash table functions come in several varieties, according to the equality test used for the keys. Plain `hash-` functions use `equal?`, `hashq-` functions use `eq?`, `hashv-` functions use `eqv?`, and the `hashx-` functions use an application supplied test.

A single `make-hash-table` creates a hash table suitable for use with any set of functions, but it’s imperative that just one set is then used consistently, or results will be unpredictable.

Hash tables are implemented as a vector indexed by a hash value formed from the key, with an association list of key/value pairs for each bucket in case distinct keys hash together. Direct access to the pairs in those lists is provided by the `*handle*` functions.

When the number of entries in a hash table goes above a threshold, the vector is made larger and the entries are rehashed, to prevent the bucket lists from becoming too long and slowing down accesses. When the number of entries goes below a threshold, the vector is shrunk to save space.

For the `hashx-` “extended” routines, an application supplies a `hash` function producing an integer index like `hashq` etc below, and an `assoc` alist search function like `assq` etc (see Section 6.6.20.3 [Retrieving Alist Entries], page 232). Here’s an example of such functions implementing case-insensitive hashing of string keys,

```lisp
(use-modules (srfi srfi-1)
```
(srFi srfi-13))

(define (my-hash str size)
  (remainder (string-hash-ci str) size))

(define (my-assoc str alist)
  (find (lambda (pair) (string-ci=? str (car pair))) alist))

(define my-table (make-hash-table))
(hashx-set! my-hash my-assoc my-table "foo" 123)

(hashx-ref my-hash my-assoc my-table "FOO")
⇒ 123

In a hashx-hash function the aim is to spread keys across the vector, so bucket lists
don't become long. But the actual values are arbitrary as long as they're in the range
0 to size – 1. Helpful functions for forming a hash value, in addition to hashq etc be-
low, include symbol-hash (see Section 6.6.6.2 [Symbol Keys], page 166), string-hash and
string-hash-ci (see Section 6.6.5.7 [String Comparison], page 148), and char-set-hash
(see Section 6.6.4.1 [Character Set Predicates/Comparison], page 134).

(make-hash-table [size]) [Scheme Procedure]
Create a new hash table object, with an optional minimum vector size.

When size is given, the table vector will still grow and shrink automatically, as de-
scribed above, but with size as a minimum. If an application knows roughly how
many entries the table will hold then it can use size to avoid rehashing when initial
entries are added.

(alist->hash-table alist) [Scheme Procedure]
alist->hashq-table alist [Scheme Procedure]
alist->hashv-table alist [Scheme Procedure]
alist->hashx-table hash assoc alist [Scheme Procedure]
Convert alist into a hash table. When keys are repeated in alist, the leftmost associ-
ation takes precedence.

(use-modules (ice-9 hash-table))
(alist->hash-table '((foo . 1) (bar . 2)))

When converting to an extended hash table, custom hash and assoc procedures must
be provided.

(alist->hashx-table hash assoc '((foo . 1) (bar . 2)))

(hash-table? obj) [Scheme Procedure]
scm_hash_table_p (obj) [C Function]
Return #t if obj is a abstract hash table object.

(hash-clear! table) [Scheme Procedure]
scm_hash_clear_x (table) [C Function]
Remove all items from table (without triggering a resize).
hash-ref table key [dflt]  
[Scheme Procedure]
hashq-ref table key [dflt]  
[Scheme Procedure]
hashv-ref table key [dflt]  
[Scheme Procedure]
hashx-ref hash assoc table key [dflt]  
[Scheme Procedure]
scm_hash_ref (table, key, dflt)  
[C Function]
scm_hashq_ref (table, key, dflt)  
[C Function]
scm_hashv_ref (table, key, dflt)  
[C Function]
scm_hashx_ref (hash, assoc, table, key, dflt)  
[C Function]

Lookup key in the given hash table, and return the associated value. If key is not found, return dflt, or #f if dflt is not given.

hash-set! table key val  
[Scheme Procedure]
hashq-set! table key val  
[Scheme Procedure]
hashv-set! table key val  
[Scheme Procedure]
hashx-set! hash assoc table key val  
[Scheme Procedure]
scm_hash_set_x (table, key, val)  
[C Function]
scm_hashq_set_x (table, key, val)  
[C Function]
scm_hashv_set_x (table, key, val)  
[C Function]
scm_hashx_set_x (hash, assoc, table, key, val)  
[C Function]

Associate val with key in the given hash table. If key is already present then it’s associated value is changed. If it’s not present then a new entry is created.

hash-remove! table key  
[Scheme Procedure]
hashq-remove! table key  
[Scheme Procedure]
hashv-remove! table key  
[Scheme Procedure]
hashx-remove! hash assoc table key  
[Scheme Procedure]
scm_hash_remove_x (table, key)  
[C Function]
scm_hashq_remove_x (table, key)  
[C Function]
scm_hashv_remove_x (table, key)  
[C Function]
scm_hashx_remove_x (hash, assoc, table, key)  
[C Function]

Remove any association for key in the given hash table. If key is not in table then nothing is done.

hash key size  
[Scheme Procedure]
hashq key size  
[Scheme Procedure]
hashv key size  
[Scheme Procedure]
scm_hash (key, size)  
[C Function]
scm_hashq (key, size)  
[C Function]
scm_hashv (key, size)  
[C Function]

Return a hash value for key. This is a number in the range 0 to size – 1, which is suitable for use in a hash table of the given size.

Note that hashq and hashv may use internal addresses of objects, so if an object is garbage collected and re-created it can have a different hash value, even when the two are notionally eq?. For instance with symbols,

```
(hashq 'something 123)  ⇒ 19
(gc)
(hashq 'something 123)  ⇒ 62
```
In normal use this is not a problem, since an object entered into a hash table won’t be garbage collected until removed. It’s only if hashing calculations are somehow separated from normal references that its lifetime needs to be considered.

### Scheme Procedure

- `hash-get-handle table key`
- `hashq-get-handle table key`
- `hashv-get-handle table key`
- `hashx-get-handle hash assoc table key`
- `scm_hash_get_handle (table, key)`
- `scm_hashq_get_handle (table, key)`
- `scm_hashv_get_handle (table, key)`
- `scm_hashx_get_handle (hash, assoc, table, key)`

Return the `(key . value)` pair for `key` in the given hash table, or `#f` if `key` is not in `table`.

### Scheme Procedure

- `hash-create-handle! table key init`
- `hashq-create-handle! table key init`
- `hashv-create-handle! table key init`
- `hashx-create-handle! hash assoc table key init`
- `scm_hash_create_handle_x (table, key, init)`
- `scm_hashq_create_handle_x (table, key, init)`
- `scm_hashv_create_handle_x (table, key, init)`
- `scm_hashx_create_handle_x (hash, assoc, table, key, init)`

Return the `(key . value)` pair for `key` in the given hash table. If `key` is not in `table` then create an entry for it with `init` as the value, and return that pair.

### Scheme Procedure

- `hash-map->list proc table`
- `hash-for-each proc table`
- `scm_hash_map_to_list (proc, table)`
- `scm_hash_for_each (proc, table)`

Apply `proc` to the entries in the given hash table. Each call is `(proc key value)`. `hash-map->list` returns a list of the results from these calls, `hash-for-each` discards the results and returns an unspecified value.

Calls are made over the table entries in an unspecified order, and for `hash-map->list` the order of the values in the returned list is unspecified. Results will be unpredictable if `table` is modified while iterating.

For example the following returns a new alist comprising all the entries from `mytable`, in no particular order.

```scheme
(hash-map->list cons mytable)
```

### Scheme Procedure

- `hash-for-each-handle proc table`
- `scm_hash_for_each_handle (proc, table)`

Apply `proc` to the entries in the given hash table. Each call is `(proc handle)`, where `handle` is a `(key . value)` pair. Return an unspecified value.

`hash-for-each-handle` differs from `hash-for-each` only in the argument list of `proc`. 
**hash-fold** proc init table  

**scm_hash_fold** (proc, init, table)  

Accumulate a result by applying proc to the elements of the given hash table. Each call is (proc key value prior-result), where key and value are from the table and prior-result is the return from the previous proc call. For the first call, prior-result is the given init value.

Calls are made over the table entries in an unspecified order. Results will be unpredictable if table is modified while hash-fold is running.

For example, the following returns a count of how many keys in mytable are strings.

```
(hash-fold (lambda (key value prior)
            (if (string? key) (1+ prior) prior))
          0 mytable)
```

**hash-count** pred table  

**scm_hash_count** (pred, table)  

Return the number of elements in the given hash table that cause (pred key value) to return true. To quickly determine the total number of elements, use (const #t) for pred.

### 6.6.23 Other Types

Procedures are documented in their own section. See Section 6.9 [Procedures], page 248.

Variable objects are documented as part of the description of Guile’s module system: see Section 6.20.7 [Variables], page 415.

See Section 6.22 [Scheduling], page 436, for discussion of threads, mutexes, and so on.

Ports are described in the section on I/O: see Section 6.14 [Input and Output], page 328.

Regular expressions are described in their own section: see Section 6.15 [Regular Expressions], page 355.

There are quite a number of additional data types documented in this manual; if you feel a link is missing here, please file a bug.

### 6.7 Foreign Objects

This chapter contains reference information related to defining and working with foreign objects. See Section 5.5 [Defining New Foreign Object Types], page 74, for a tutorial-like introduction to foreign objects.

**scm_t_struct_finalize**  

This function type returns void and takes one SCM argument.

**SCM scm_make_foreign_object_type** (SCM name, SCM slots, scm_t_struct_finalize finalizer)  

Create a fresh foreign object type. name is a symbol naming the type. slots is a list of symbols, each one naming a field in the foreign object type. finalizer indicates the finalizer, and may be NULL.
We recommend that finalizers be avoided if possible. See Section 5.5.4 [Foreign Object Memory Management], page 77. Finalizers must be async-safe and thread-safe. Again, see Section 5.5.4 [Foreign Object Memory Management], page 77. If you are embedding Guile in an application that is not thread-safe, and you define foreign object types that need finalization, you might want to disable automatic finalization, and arrange to call \texttt{scm\_manually\_run\_finalizers()} yourself.

\texttt{int \textbf{scm\_set\_automatic\_finalization\_enabled}(int \texttt{enabled\_p})} \quad \text{[C Function]}

Enable or disable automatic finalization. By default, Guile arranges to invoke object finalizers automatically, in a separate thread if possible. Passing a zero value for \texttt{enabled\_p} will disable automatic finalization for Guile as a whole. If you disable automatic finalization, you will have to call \texttt{scm\_run\_finalizers()} periodically.

Unlike most other Guile functions, you can call \texttt{scm\_set\_automatic\_finalization\_enabled()} before Guile has been initialized.

Return the previous status of automatic finalization.

\texttt{int \textbf{scm\_run\_finalizers}(void)} \quad \text{[C Function]}

Invoke any pending finalizers. Returns the number of finalizers that were invoked. This function should be called when automatic finalization is disabled, though it may be called if it is enabled as well.

\texttt{void \textbf{scm\_assert\_foreign\_object\_type}(SCM \texttt{type}, SCM \texttt{val})} \quad \text{[C Function]}

When \texttt{val} is a foreign object of the given \texttt{type}, do nothing. Otherwise, signal an error.

\texttt{SCM \textbf{scm\_make\_foreign\_object\_0}(SCM \texttt{type})} \quad \text{[C Function]}

\texttt{SCM \textbf{scm\_make\_foreign\_object\_1}(SCM \texttt{type}, void *\texttt{val0})} \quad \text{[C Function]}

\texttt{SCM \textbf{scm\_make\_foreign\_object\_2}(SCM \texttt{type}, void *\texttt{val0}, void *\texttt{val1})} \quad \text{[C Function]}

\texttt{SCM \textbf{scm\_make\_foreign\_object\_3}(SCM \texttt{type}, void *\texttt{val0}, void *\texttt{val1}, void *\texttt{val2})} \quad \text{[C Function]}

\texttt{SCM \textbf{scm\_make\_foreign\_object\_n}(SCM \texttt{type}, size\_t \texttt{n}, void *\texttt{vals}[])} \quad \text{[C Function]}

Make a new foreign object of the type with type \texttt{type} and initialize the first \texttt{n} fields to the given values, as appropriate.

The number of fields for objects of a given type is fixed when the type is created. It is an error to give more initializers than there are fields in the value. It is perfectly fine to give fewer initializers than needed; this is convenient when some fields are of non-pointer types, and would be easier to initialize with the setters described below.

\texttt{void* \textbf{scm\_foreign\_object\_ref}(SCM \texttt{obj}, size\_t \texttt{n});} \quad \text{[C Function]}

\texttt{scm\_t\_bits \textbf{scm\_foreign\_object\_unsigned\_ref}(SCM \texttt{obj}, size\_t \texttt{n});} \quad \text{[C Function]}

\texttt{scm\_t\_signed\_bits \textbf{scm\_foreign\_object\_signed\_ref}(SCM \texttt{obj}, size\_t \texttt{n});} \quad \text{[C Function]}

Return the value of the \texttt{n}th field of the foreign object \texttt{obj}. The backing store for the fields is as wide as a \texttt{scm\_t\_bits} value, which is at least as wide as a pointer. The different variants handle casting in a portable way.
void scm_foreign_object_set_x (SCM obj, size_t n, void *val);  
[C Function]
void scm_foreign_object_unsigned_set_x (SCM obj, size_t n, 
scm_t_bits val);  
[C Function]
void scm_foreign_object_signed_set_x (SCM obj, size_t n, 
scm_t_signed_bits val);  

Set the value of the nth field of the foreign object obj to val, after portably converting 
to a scm_t_bits value, if needed.

One can also access foreign objects from Scheme. See Section 5.5.5 [Foreign Objects and 
Scheme], page 80, for some examples.

(use-modules (system foreign-object))

make-foreign-object-type name slots [#:finalizer=\#f]  
[Scheme Procedure]
Make a new foreign object type. See the above documentation for scm_make_ 
foreign_object_type; these functions are exactly equivalent, except for the way in 
which the finalizer gets attached to instances (an internal detail).

The resulting value is a GOOPS class. See Chapter 8 [GOOPS], page 759, for more 
on classes in Guile.

define-foreign-object-type name constructor (slot ...)  
[Scheme Syntax]  
[#:finalizer=\#f]
A convenience macro to define a type, using make-foreign-object-type, and bind 
it to name. A constructor will be bound to constructor, and getters will be bound to 
each of slot....

6.8 Smobs

A smob is a “small object”. Before foreign objects were introduced in Guile 2.0.12 (see 
Section 6.7 [Foreign Objects], page 243), smobs were the preferred way to for C code to 
define new kinds of Scheme objects. With the exception of the so-called “applicable SMOBs” discussed 
below, smobs are now a legacy interface and are headed for eventual deprecation. See Section 6.2 [Deprecation], page 100. New code should use the foreign object interface.

This section contains reference information related to defining and working with smobs. For a tutorial-like introduction to smobs, see “Defining New Types (Smobs)” in previous 
versions of this manual.

scm_t_bits scm_make_smob_type (const char *name, size_t size)  
[Function]
This function adds a new smob type, named name, with instance size size, to the 
system. The return value is a tag that is used in creating instances of the type.

If size is 0, the default free function will do nothing.

If size is not 0, the default free function will deallocate the memory block pointed to by 
SCM_SMOB_DATA with scm_gc_free. The what parameter in the call to scm_gc_free 
will be name.

Default values are provided for the mark, free, print, and equalp functions. If you 
want to customize any of these functions, the call to scm_make_smob_type should 
be immediately followed by calls to one or several of scm_set_smob_mark, scm_set_ 
smob_free, scm_set_smob_print, and/or scm_set_smob_equalp.
void scm_set_smob_free (scm_t_bits tc, size_t (*free) (SCM obj))  [C Function]
This function sets the smob freeing procedure (sometimes referred to as a finalizer) for the smob type specified by the tag tc. tc is the tag returned by scm_make_smob_type.
The free procedure must deallocate all resources that are directly associated with the smob instance obj. It must assume that all SCM values that it references have already been freed and are thus invalid.
It must also not call any libguile function or macro except scm_gc_free, SCM_SMOB_FLAGS, SCM_SMOB_DATA, SCM_SMOB_DATA_2, and SCM_SMOB_DATA_3.
The free procedure must return 0.
Note that defining a freeing procedure is not necessary if the resources associated with obj consists only of memory allocated with scm_gc_malloc or scm_gc_malloc_pointerless because this memory is automatically reclaimed by the garbage collector when it is no longer needed (see Section 6.19.2 [Memory Blocks], page 402).

Smob free functions must be thread-safe. See Section 5.5.4 [Foreign Object Memory Management], page 77, for a discussion on finalizers and concurrency. If you are embedding Guile in an application that is not thread-safe, and you define smob types that need finalization, you might want to disable automatic finalization, and arrange to call scm_manually_run_finalizers () yourself. See Section 6.7 [Foreign Objects], page 243.

void scm_set_smob_mark (scm_t_bits tc, SCM (*mark) (SCM obj))  [C Function]
This function sets the smob marking procedure for the smob type specified by the tag tc. tc is the tag returned by scm_make_smob_type.
Defining a marking procedure is almost always the wrong thing to do. It is much, much preferable to allocate smob data with the scm_gc_malloc and scm_gc_malloc_pointerless functions, and allow the GC to trace pointers automatically.
Any mark procedures you see currently almost surely date from the time of Guile 1.8, before the switch to the Boehm-Demers-Weiser collector. Such smob implementations should be changed to just use scm_gc_malloc and friends, and to lose their mark function.
If you decide to keep the mark function, note that it may be called on objects that are on the free list. Please read and digest the comments from the BDW GC’s gc/gc_mark.h header.
The mark procedure must cause scm_gc_mark to be called for every SCM value that is directly referenced by the smob instance obj. One of these SCM values can be returned from the procedure and Guile will call scm_gc_mark for it. This can be used to avoid deep recursions for smob instances that form a list.
It must not call any libguile function or macro except scm_gc_mark, SCM_SMOB_FLAGS, SCM_SMOB_DATA, SCM_SMOB_DATA_2, and SCM_SMOB_DATA_3.

void scm_set_smob_print (scm_t_bits tc, int (*print) (SCM obj, SCM port, scm_print_state* pstate))  [C Function]
This function sets the smob printing procedure for the smob type specified by the tag tc. tc is the tag returned by scm_make_smob_type.
The print procedure should output a textual representation of the smob instance obj to port, using information in pstate.

The textual representation should be of the form #<name ...>. This ensures that read will not interpret it as some other Scheme value.

It is often best to ignore pstate and just print to port with scm_display, scm_write, scm_simple_format, and scm_puts.

```c
void scm_set_smob_equalp (scm_t_bits tc, SCM (*equalp) (SCM obj1, SCM obj2))

This function sets the smob equality-testing predicate for the smob type specified by the tag tc. tc is the tag returned by scm_make_smob_type.

The equalp procedure should return SCM_BOOL_T when obj1 is equal? to obj2. Else it should return SCM_BOOL_F. Both obj1 and obj2 are instances of the smob type tc.
```

```c
void scm_assert_smob_type (scm_t_bits tag, SCM val)  

When val is a smob of the type indicated by tag, do nothing. Else, signal an error.
```

```c
int SCM_SMOB_PREDICATE (scm_t_bits tag, SCM exp)

Return true if exp is a smob instance of the type indicated by tag, or false otherwise. The expression exp can be evaluated more than once, so it shouldn’t contain any side effects.
```

```c
SCM scm_new_smob (scm_t_bits tag, void *data)  
SCM scm_new_double_smob (scm_t_bits tag, void *data, void *data2, void *data3)

Make a new smob of the type with tag tag and smob data data, data2, and data3, as appropriate.

The tag is what has been returned by scm_make_smob_type. The initial values data, data2, and data3 are of type scm_t_bits; when you want to use them for SCM values, these values need to be converted to a scm_t_bits first by using SCM_UNPACK.

The flags of the smob instance start out as zero.

```c
scm_t_bits SCM_SMOB_FLAGS (SCM obj)

Return the 16 extra bits of the smob obj. No meaning is predefined for these bits, you can use them freely.
```

```c
scm_t_bits SCM_SET_SMOB_FLAGS (SCM obj, scm_t_bits flags)

Set the 16 extra bits of the smob obj to flags. No meaning is predefined for these bits, you can use them freely.
```

```c
scm_t_bits SCM_SMOB_DATA (SCM obj)  
scm_t_bits SCM_SMOB_DATA_2 (SCM obj)  
scm_t_bits SCM_SMOB_DATA_3 (SCM obj)

Return the first (second, third) immediate word of the smob obj as a scm_t_bits value. When the word contains a SCM value, use SCM_SMOB_OBJECT (etc.) instead.
```

```c
void SCM_SET_SMOB_DATA (SCM obj, scm_t_bits val)  
void SCM_SET_SMOB_DATA_2 (SCM obj, scm_t_bits val)
```

void SCM_SET_SMOB_DATA_3 (SCM obj, scm_t_bits val)  
Set the first (second, third) immediate word of the smob obj to val. When the word 
should be set to a SCM value, use SCM_SMOB_SET_OBJECT (etc.) instead.

SCM SCM_SMOB_OBJECT (SCM obj)  
Return the first (second, third) immediate word of the smob obj as a SCM value. When 
the word contains a scm_t_bits value, use SCM_SMOB_DATA (etc.) instead.

void SCM_SET_SMOB_OBJECT (SCM obj, SCM val)  
void SCM_SET_SMOB_OBJECT_2 (SCM obj, SCM val)  
void SCM_SET_SMOB_OBJECT_3 (SCM obj, SCM val)  
Set the first (second, third) immediate word of the smob obj to val. When the word 
should be set to a scm_t_bits value, use SCM_SMOB_SET_DATA (etc.) instead.

SCM * SCM_SMOB_OBJECT_LOC (SCM obj)  
SCM * SCM_SMOB_OBJECT_2_LOC (SCM obj)  
SCM * SCM_SMOB_OBJECT_3_LOC (SCM obj)  
Return a pointer to the first (second, third) immediate word of the smob obj. Note 
that this is a pointer to SCM. If you need to work with scm_t_bits values, use 
SCM_PACK and SCM_UNPACK, as appropriate.

SCM scm_markcdr (SCM x)  
Mark the references in the smob x, assuming that x’s first data word contains an 
ordinary Scheme object, and x refers to no other objects. This function simply 
returns x’s first data word.

6.9 Procedures

6.9.1 Lambda: Basic Procedure Creation

A lambda expression evaluates to a procedure. The environment which is in effect when a 
lambda expression is evaluated is enclosed in the newly created procedure, this is referred 
to as a closure (see Section 3.4 [About Closure], page 26).

When a procedure created by lambda is called with some actual arguments, the environ-
ment enclosed in the procedure is extended by binding the variables named in the formal 
argument list to new locations and storing the actual arguments into these locations. Then 
the body of the lambda expression is evaluated sequentially. The result of the last expression 
in the procedure body is then the result of the procedure invocation.

The following examples will show how procedures can be created using lambda, and 
what you can do with these procedures.

(lambda (x) (+ x x))  ⇒  a procedure
(((lambda (x) (+ x x)) 4)  ⇒  8

The fact that the environment in effect when creating a procedure is enclosed in the 
procedure is shown with this example:

(define add4
(let ((x 4))
  (lambda (y) (+ x y)))
(add4 6) ⇒ 10

**lambda formals body** [syntax]
formals should be a formal argument list as described in the following table.

\[(variable1 \ldots)\]
The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored into the newly created location for the formal variables.

\[variable\]
The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments will converted into a list and stored into the newly created location for the formal variable.

\[(variable1 \ldots variablen . variablen+1)\]
If a space-delimited period precedes the last variable, then the procedure takes \(n\) or more variables where \(n\) is the number of formal arguments before the period. There must be at least one argument before the period. The first \(n\) actual arguments will be stored into the newly allocated locations for the first \(n\) formal arguments and the sequence of the remaining actual arguments is converted into a list and the stored into the location for the last formal argument. If there are exactly \(n\) actual arguments, the empty list is stored into the location of the last formal argument.

The list in variable or variablen+1 is always newly created and the procedure can modify it if desired. This is the case even when the procedure is invoked via apply, the required part of the list argument there will be copied (see Section 6.18.4 [Procedures for On the Fly Evaluation], page 384).

\[body\] is a sequence of Scheme expressions which are evaluated in order when the procedure is invoked.

### 6.9.2 Primitive Procedures

Procedures written in C can be registered for use from Scheme, provided they take only arguments of type SCM and return SCM values. scm_c_define_gsubr is likely to be the most useful mechanism, combining the process of registration (scm_c_make_gsubr) and definition (scm_define).

**SCM scm_c_make_gsubr** (const char *name, int req, int opt, int rst, fcn) [Function]

Register a C procedure \(fcn\) as a “subr” — a primitive subroutine that can be called from Scheme. It will be associated with the given name but no environment binding will be created. The arguments req, opt and rst specify the number of required, optional and “rest” arguments respectively. The total number of these arguments should match the actual number of arguments to \(fcn\), but may not exceed 10. The number of rest arguments should be 0 or 1. scm_c_make_gsubr returns a value of type SCM which is a “handle” for the procedure.
SCM scm_c_define_gsubr (const char *name, int req, int opt, int rst, fcn)  
Register a C procedure fcn, as for scm_c_make_gsubr above, and additionally create a top-level Scheme binding for the procedure in the “current environment” using scm_define. scm_c_define_gsubr returns a handle for the procedure in the same way as scm_c_make_gsubr, which is usually not further required.

6.9.3 Compiled Procedures
The evaluation strategy given in Section 6.9.1 [Lambda], page 248, describes how procedures are interpreted. Interpretation operates directly on expanded Scheme source code, recursively calling the evaluator to obtain the value of nested expressions.

Most procedures are compiled, however. This means that Guile has done some pre-computation on the procedure, to determine what it will need to do each time the procedure runs. Compiled procedures run faster than interpreted procedures.

Loading files is the normal way that compiled procedures come to being. If Guile sees that a file is uncompiled, or that its compiled file is out of date, it will attempt to compile the file when it is loaded, and save the result to disk. Procedures can be compiled at runtime as well. See Section 6.18 [Read/Load/Eval/Compile], page 379, for more information on runtime compilation.

Compiled procedures, also known as programs, respond all procedures that operate on procedures. In addition, there are a few more accessors for low-level details on programs.

Most people won’t need to use the routines described in this section, but it’s good to have them documented. You’ll have to include the appropriate module first, though:

(use-modules (system vm program))

program? obj
scm_program_p (obj)
Returns #t if obj is a compiled procedure, or #f otherwise.

program-code program
scm_program_code (program)
Returns the address of the program’s entry, as an integer. This address is mostly useful to procedures in (system vm debug).

program-num-free-variable program
scm_program_num_free_variables (program)
Return the number of free variables captured by this program.

program-free-variable-ref program n
scm_program_free_variable-ref (program, n)
program-free-variable-set! program n val
scm_program_free_variable_set_x (program, n, val)
Accessors for a program’s free variables. Some of the values captured are actually in variable “boxes”. See Section 9.3.4 [Variables and the VM], page 815, for more information.

Users must not modify the returned value unless they think they’re really clever.
Bindings annotations for programs, along with their accessors.

Bindings declare names and liveness extents for block-local variables. The best way to see what these are is to play around with them at a REPL. See Section 9.3.2 [VM Concepts], page 813, for more information.

Note that bindings information is stored in a program as part of its metadata thunk, so including it in the generated object code does not impose a runtime performance penalty.

Source location annotations for programs, along with their accessors.

Source location information propagates through the compiler and ends up being serialized to the program’s metadata. This information is keyed by the offset of the instruction pointer within the object code of the program. Specifically, it is keyed on the ip just following an instruction, so that backtraces can find the source location of a call that is in progress.

Accessors for a representation of the “arity” of a program.

The normal case is that a procedure has one arity. For example, \((\text{lambda (x) x})\), takes one required argument, and that’s it. One could access that number of required arguments via \((\text{arity:nreq (program-arities (lambda (x) x))})\). Similarly, \(\text{arity:nopt}\) gets the number of optional arguments, and \(\text{arity:rest?}\) returns a true value if the procedure has a rest arg.

\(\text{arity:kw}\) returns a list of \((\text{kw . idx})\) pairs, if the procedure has keyword arguments. The \(\text{idx}\) refers to the \(\text{idx}\)th local variable; See Section 9.3.4 [Variables and the VM], page 815, for more information. Finally \(\text{arity:allow-other-keys?}\) returns a true
value if other keys are allowed. See Section 6.9.4 [Optional Arguments], page 252, for
more information.

So what about arity:start and arity:end, then? They return the range of bytes
in the program’s bytecode for which a given arity is valid. You see, a procedure can
actually have more than one arity. The question, “what is a procedure’s arity” only
really makes sense at certain points in the program, delimited by these arity:start
and arity:end values.

**program-arguments-alist** *program* [ip]

[Scheme Procedure]
Return an association list describing the arguments that *program* accepts, or #f if
the information cannot be obtained.

The alist keys that are currently defined are ‘required’, ‘optional’, ‘keyword’, ‘allow-
other-keys?’, and ‘rest’. For example:

```
(program-arguments-alist
 (lambda* (a b #:optional c #:key (d 1) #:rest e)
  #t)) ⇒
 ((required . (a b))
  (optional . (c))
  (keyword . ((#:d . 4)))
  (allow-other-keys? . #f)
  (rest . d))
```

**program-lambda-list** *program* [ip]

[Scheme Procedure]
Return a representation of the arguments of *program* as a lambda list, or #f if this
information is not available.

For example:

```
(program-lambda-list
 (lambda* (a b #:optional c #:key (d 1) #:rest e)
  #t)) ⇒
```

### 6.9.4 Optional Arguments

Scheme procedures, as defined in R5RS, can either handle a fixed number of actual
arguments, or a fixed number of actual arguments followed by arbitrarily many additional
arguments. Writing procedures of variable arity can be useful, but unfortunately, the syn-
tactic means for handling argument lists of varying length is a bit inconvenient. It is possible
to give names to the fixed number of arguments, but the remaining (optional) arguments
can be only referenced as a list of values (see Section 6.9.1 [Lambda], page 248).

For this reason, Guile provides an extension to lambda, lambda*, which allows the user
to define procedures with optional and keyword arguments. In addition, Guile’s virtual
machine has low-level support for optional and keyword argument dispatch. Calls to pro-
cedures with optional and keyword arguments can be made cheaply, without allocating a
rest list.

### 6.9.4.1 lambda* and define*.

lambda* is like lambda, except with some extensions to allow optional and keyword argu-
ments.
Create a procedure which takes optional and/or keyword arguments specified with #:optional and #:key. For example,

    (lambda* (a b #:optional c d . e) '())

is a procedure with fixed arguments \(a\) and \(b\), optional arguments \(c\) and \(d\), and rest argument \(e\). If the optional arguments are omitted in a call, the variables for them are bound to #f.

Likewise, define* is syntactic sugar for defining procedures using lambda*.

lambda* can also make procedures with keyword arguments. For example, a procedure defined like this:

    (define* (sir-yes-sir #:key action how-high)
      (list action how-high))

can be called as (sir-yes-sir #:action 'jump), (sir-yes-sir #:how-high 13), (sir-yes-sir #:action 'lay-down #:how-high 0), or just (sir-yes-sir). Whichever arguments are given as keywords are bound to values (and those not given are #f).

Optional and keyword arguments can also have default values to take when not present in a call, by giving a two-element list of variable name and expression. For example in

    (define* (frob foo #:optional (bar 42) #:key (baz 73))
      (list foo bar baz))

\(foo\) is a fixed argument, \(bar\) is an optional argument with default value 42, and \(baz\) is a keyword argument with default value 73. Default value expressions are not evaluated unless they are needed, and until the procedure is called.

Normally it’s an error if a call has keywords other than those specified by #:key, but adding #:allow-other-keys to the definition (after the keyword argument declarations) will ignore unknown keywords.

If a call has a keyword given twice, the last value is used. For example,

    (define* (flips #:key (heads 0) (tails 0))
      (display (list heads tails)))

    (flips #:heads 37 #:tails 42 #:heads 99)

    ⊣ (99 42)

#:rest is a synonym for the dotted syntax rest argument. The argument lists (a . b) and (a #:rest b) are equivalent in all respects. This is provided for more similarity to DSSSL, MIT-Scheme and Kawa among others, as well as for refugees from other Lisp dialects.

When #:key is used together with a rest argument, the keyword parameters in a call all remain in the rest list. This is the same as Common Lisp. For example,
((lambda* (#:key (x 0) #:allow-other-keys #:rest r)
  (display r))
  #:x 123 #:y 456)
⊣
  (#:x 123 #:y 456)

#:optional and #:key establish their bindings successively, from left to right. This means default expressions can refer back to prior parameters, for example

(l lambda* (start #:optional (end (+ 10 start)))
  (do ((i start (1+ i)))
      ((> i end))
      (display i)))

The exception to this left-to-right scoping rule is the rest argument. If there is a rest argument, it is bound after the optional arguments, but before the keyword arguments.

6.9.4.2 (ice-9 optargs)

Before Guile 2.0, lambda* and define* were implemented using macros that processed rest list arguments. This was not optimal, as calling procedures with optional arguments had to allocate rest lists at every procedure invocation. Guile 2.0 improved this situation by bringing optional and keyword arguments into Guile’s core.

However there are occasions in which you have a list and want to parse it for optional or keyword arguments. Guile’s (ice-9 optargs) provides some macros to help with that task.

The syntax let-optional and let-optional* are for destructuring rest argument lists and giving names to the various list elements. let-optional binds all variables simultaneously, while let-optional* binds them sequentially, consistent with let and let* (see Section 6.12.2 [Local Bindings], page 295).

let-optional  rest-arg  (binding . . .)  body1  body2  . . .  [library syntax]
let-optional* rest-arg  (binding . . .)  body1  body2  . . .  [library syntax]

These two macros give you an optional argument interface that is very Schemey and introduces no fancy syntax. They are compatible with the scsh macros of the same name, but are slightly extended. Each of binding may be of one of the forms var or (var default-value). rest-arg should be the rest-argument of the procedures these are used from. The items in rest-arg are sequentially bound to the variable names are given. When rest-arg runs out, the remaining vars are bound either to the default values or #f if no default value was specified. rest-arg remains bound to whatever may have been left of rest-arg.

After binding the variables, the expressions body1 body2 . . . are evaluated in order.

Similarly, let-keywords and let-keywords* extract values from keyword style argument lists, binding local variables to those values or to defaults.
let-keywords args allow-other-keys? (binding . . . ) body1 [library syntax] 

body2 . . . 

let-keywords* args allow-other-keys? (binding . . . ) body1 [library syntax] 

body2 . . . 

args is evaluated and should give a list of the form (#:keyword1 value1 #:keyword2 value2 . . . ). The bindings are variables and default expressions, with the variables to be set (by name) from the keyword values. The body1 body2 . . . forms are then evaluated and the last is the result. An example will make the syntax clearest, 

(define args '(#:xyzzy "hello" #:foo "world") )

(let-keywords args #t 

  (foo "default for foo"
  
  (bar (string-append "default" " for" " bar")))

  (display foo)
  (display ", ")
  (display bar))

world, defaultforbar

The binding for foo comes from the #:foo keyword in args. But the binding for bar is the default in the let-keywords, since there’s no #:bar in the args. 

allow-other-keys? is evaluated and controls whether unknown keywords are allowed in the args list. When true other keys are ignored (such as #:xyzzy in the example), when #f an error is thrown for anything unknown.

(ice-9 optargs) also provides some more define* sugar, which is not so useful with modern Guile coding, but still supported: define*-public is the lambda* version of define-public; defmacro* and defmacro*-public exist for defining macros with the improved argument list handling possibilities. The -public versions not only define the procedures/macros, but also export them from the current module.

define*-public formals body1 body2 . . . [library syntax] 

Like a mix of define* and define-public.

defmacro* name formals body1 body2 . . . [library syntax] 
defmacro*-public name formals body1 body2 . . . [library syntax] 

These are just like defmacro and defmacro-public except that they take lambda*-style extended parameter lists, where #:optional, #:key, #:allow-other-keys and #:rest are allowed with the usual semantics. Here is an example of a macro with an optional argument:

(defmacro* transmogrify (a #:optional b) 

(a 1))

6.9.5 Case-lambda

R5RS’s rest arguments are indeed useful and very general, but they often aren’t the most appropriate or efficient means to get the job done. For example, lambda* is a much better solution to the optional argument problem than lambda with rest arguments.

Likewise, case-lambda works well for when you want one procedure to do double duty (or triple, or ...), without the penalty of consing a rest list.
For example:

```scheme
(define (make-accum n)
  (case-lambda
    (() n)
    ((m) (set! n (+ n m)) n)))

(define a (make-accum 20))
(a) ⇒ 20
(a 10) ⇒ 30
(a) ⇒ 30
```

The value returned by a `case-lambda` form is a procedure which matches the number of actual arguments against the formals in the various clauses, in order. The first matching clause is selected, the corresponding values from the actual parameter list are bound to the variable names in the clauses and the body of the clause is evaluated. If no clause matches, an error is signalled.

The syntax of the `case-lambda` form is defined in the following EBNF grammar. *Formals* means a formal argument list just like with `lambda` (see Section 6.9.1 [Lambda], page 248).

```
<case-lambda>
  --> (case-lambda <case-lambda-clause>*
  --> (case-lambda <docstring> <case-lambda-clause>*)
<case-lambda-clause>
  --> (<formals> <definition-or-command>*
<formals>
  --> (<identifier>*)
  | (<identifier>* . <identifier>)
  | <identifier>
```

Rest lists can be useful with `case-lambda`:

```scheme
(define plus
  (case-lambda
    "Return the sum of all arguments."
    (() 0)
    ((a) a)
    ((a b) (+ a b))
    ((a b . rest) (apply plus (+ a b) rest))))

(plus 1 2 3) ⇒ 6
```

Also, for completeness. Guile defines `case-lambda*` as well, which is like `case-lambda`, except with `lambda*` clauses. A `case-lambda*` clause matches if the arguments fill the required arguments, but are not too many for the optional and/or rest arguments.

Keyword arguments are possible with `case-lambda*` as well, but they do not contribute to the “matching” behavior, and their interactions with required, optional, and rest arguments can be surprising.

For the purposes of `case-lambda*` (and of `case-lambda`, as a special case), a clause *matches* if it has enough required arguments, and not too many positional arguments. The required arguments are any arguments before the `#:optional`, `#:key`, and `#:rest`
arguments. *Positional* arguments are the required arguments, together with the optional arguments.

In the absence of #:key or #:rest arguments, it’s easy to see how there could be too many positional arguments: you pass 5 arguments to a function that only takes 4 arguments, including optional arguments. If there is a #:rest argument, there can never be too many positional arguments: any application with enough required arguments for a clause will match that clause, even if there are also #:key arguments.

Otherwise, for applications to a clause with #:key arguments (and without a #:rest argument), a clause will match there only if there are enough required arguments and if the next argument after binding required and optional arguments, if any, is a keyword. For efficiency reasons, Guile is currently unable to include keyword arguments in the matching algorithm. Clauses match on positional arguments only, not by comparing a given keyword to the available set of keyword arguments that a function has.

Some examples follow.

```
(define f
  (case-lambda*
    ((a #:optional b) 'clause-1)
    ((a #:optional b #:key c) 'clause-2)
    ((a #:key d) 'clause-3)
    ((#:key e #:rest f) 'clause-4)))

(f) ⇒ clause-4
(f 1) ⇒ clause-1
(f) ⇒ clause-4
(f #:e 10) clause-1
(f 1 #:foo) clause-1
(f 1 #:c 2) clause-2
(f #:a #:b #:c #:d #:e) clause-4
```

;; clause-2 will match anything that clause-3 would match.
(f 1 #:d 2) ⇒ error: bad keyword args in clause 2

Don’t forget that the clauses are matched in order, and the first matching clause will be taken. This can result in a keyword being bound to a required argument, as in the case of f #:e 10.

### 6.9.6 Higher-Order Functions

As a functional programming language, Scheme allows the definition of *higher-order functions*, i.e., functions that take functions as arguments and/or return functions. Utilities to derive procedures from other procedures are provided and described below.

#### const value

Return a procedure that accepts any number of arguments and returns value.

```
(procedure? (const 3)) ⇒ #t
((const 'hello)) ⇒ hello
((const 'hello) 'world) ⇒ hello
```
negate proc  
Return a procedure with the same arity as proc that returns the not of proc’s result.

(procedure? (negate number?)) ⇒ #t  
((negate odd?) 2) ⇒ #t  
((negate real?) 'dream) ⇒ #t  
((negate string-prefix?) "GNU" "GNU Guile") ⇒ #f  
(filter (negate number?) '(a 2 "b")) ⇒ (a "b")

compose proc1 proc2 ...  
Compose proc1 with the procedures proc2 ... such that the last proc argument is applied first and proc1 last, and return the resulting procedure. The given procedures must have compatible arity.

(procedure? (compose 1+ 1-)) ⇒ #t  
((compose sqrt 1+ 1+) 2) ⇒ 2.0  
((compose 1+ sqrt) 3) ⇒ 2.73205080756888  
(eq? (compose 1+) 1+) ⇒ #t  
((compose zip unzip2) '((1 2) (a b))) ⇒ '((1 2) (a b))

identity x  
Return X.

and=> value proc  
When value is #f, return #f. Otherwise, return (proc value).

6.9.7 Procedure Properties and Meta-information
In addition to the information that is strictly necessary to run, procedures may have other associated information. For example, the name of a procedure is information not for the procedure, but about the procedure. This meta-information can be accessed via the procedure properties interface.

The first group of procedures in this meta-interface are predicates to test whether a Scheme object is a procedure, or a special procedure, respectively. procedure? is the most general predicate, it returns #t for any kind of procedure.

procedure? obj  
scm_procedure_p (obj)  
Return #t if obj is a procedure.

thunk? obj  
scm_thunk_p (obj)  
Return #t if obj is a thunk—a procedure that does not accept arguments.

Procedure properties are general properties associated with procedures. These can be the name of a procedure or other relevant information, such as debug hints.
6.9.8 Procedures with Setters

A procedure with setter is a special kind of procedure which normally behaves like any accessor procedure, that is a procedure which accesses a data structure. The difference is that this kind of procedure has a so-called setter attached, which is a procedure for storing something into a data structure.

Procedures with setters are treated specially when the procedure appears in the special form set! (REFIXME). How it works is best shown by example.

Suppose we have a procedure called foo-ref, which accepts two arguments, a value of type foo and an integer. The procedure returns the value stored at the given index in the foo object. Let f be a variable containing such a foo data structure.

\[
\text{(foo-ref f 0)} \quad \Rightarrow \quad \text{bar}
\]

4 Working definitions would be:

\[
\begin{align*}
&\text{(define foo-ref vector-ref)} \\
&\text{(define foo-set! vector-set!)} \\
&\text{(define f (make-vector 2 #f))}
\end{align*}
\]
Also suppose that a corresponding setter procedure called \texttt{foo-set!} does exist.

\begin{verbatim}
(foo-set! f 0 'bla)
(foo-ref f 0) \Rightarrow bla
\end{verbatim}

Now we could create a new procedure called \texttt{foo}, which is a procedure with setter, by calling \texttt{make-procedure-with-setter} with the accessor and setter procedures \texttt{foo-ref} and \texttt{foo-set!}. Let us call this new procedure \texttt{foo}.

\begin{verbatim}
(define foo (make-procedure-with-setter foo-ref foo-set!))
\end{verbatim}

\texttt{foo} can from now on be used to either read from the data structure stored in \texttt{f}, or to write into the structure.

\begin{verbatim}
(set! (foo f 0) 'dum)
(foo f 0) \Rightarrow dum
\end{verbatim}

\texttt{make-procedure-with-setter} \texttt{procedure setter} \hspace{1cm} \text{[Scheme Procedure]}
\texttt{scm_make_procedure_with_setter (procedure, setter)} \hspace{1cm} \text{[C Function]}

Create a new procedure which behaves like \texttt{procedure}, but with the associated setter \texttt{setter}.

\texttt{procedure-with-setter? obj} \hspace{1cm} \text{[Scheme Procedure]}
\texttt{scm_procedure_with_setter_p (obj)} \hspace{1cm} \text{[C Function]}

Return \texttt{#t} if \texttt{obj} is a procedure with an associated setter procedure.

\texttt{procedure proc} \hspace{1cm} \text{[Scheme Procedure]}
\texttt{scm_procedure (proc)} \hspace{1cm} \text{[C Function]}

Return the procedure of \texttt{proc}, which must be an applicable struct.

\texttt{setter proc} \hspace{1cm} \text{[Scheme Procedure]}

Return the setter of \texttt{proc}, which must be either a procedure with setter or an operator struct.

### 6.9.9 Inlinable Procedures

You can define an \textit{inlinable procedure} by using \texttt{define-inlinable} instead of \texttt{define}. An inlinable procedure behaves the same as a regular procedure, but direct calls will result in the procedure body being inlined into the caller.

Bear in mind that starting from version 2.0.3, Guile has a partial evaluator that can inline the body of inner procedures when deemed appropriate:

\begin{verbatim}
scheme@(guile-user)> ,optimize (define (foo x)
    (define (bar) (+ x 3))
    (* (bar) 2))
$1 = (define foo
    (lambda (#\{x 94}\#) (* (+ #\{x 94}\# 3) 2)))
\end{verbatim}

The partial evaluator does not inline top-level bindings, though, so this is a situation where you may find it interesting to use \texttt{define-inlinable}.

Procedures defined with \texttt{define-inlinable} are \textit{always} inlined, at all direct call sites. This eliminates function call overhead at the expense of an increase in code size. Additionally, the caller will not transparently use the new definition if the inline procedure is
redefined. It is not possible to trace an inlined procedures or install a breakpoint in it (see Section 6.26.4 [Traps], page 480). For these reasons, you should not make a procedure inlinable unless it demonstrably improves performance in a crucial way.

In general, only small procedures should be considered for inlining, as making large procedures inlinable will probably result in an increase in code size. Additionally, the elimination of the call overhead rarely matters for large procedures.

\[
\text{define-inlinable (name parameter ...) body1 body2 ...} \quad \text{[Scheme Syntax]}
\]

Define \text{name} as a procedure with parameters \text{parameters} and bodies \text{body1}, \text{body2}, ...

6.10 Macros

At its best, programming in Lisp is an iterative process of building up a language appropriate to the problem at hand, and then solving the problem in that language. Defining new procedures is part of that, but Lisp also allows the user to extend its syntax, with its famous macros.

Macros are syntactic extensions which cause the expression that they appear in to be transformed in some way \textit{before} being evaluated. In expressions that are intended for macro transformation, the identifier that names the relevant macro must appear as the first element, like this:

\[
\text{(macro-name macro-args ...)}
\]

Macro expansion is a separate phase of evaluation, run before code is interpreted or compiled. A macro is a program that runs on programs, translating an embedded language into core Scheme\textsuperscript{5}.

6.10.1 Defining Macros

A macro is a binding between a keyword and a syntax transformer. Since it’s difficult to discuss \texttt{define-syntax} without discussing the format of transformers, consider the following example macro definition:

\[
\text{(define-syntax when}
\text{(syntax-rules ()}
\text{((-when condition exp ...)}
\text{\hspace{1em} (if condition}
\text{\hspace{1em} \hspace{1em} (begin exp ...)))))}
\]

\[
\text{(when #t}
\text{\hspace{1em} (display "hey ho
")}
\text{\hspace{1em} (display "let’s go
")}
\text{-| hey ho}
\text{-| let’s go}
\]

In this example, the \texttt{when} binding is bound with \texttt{define-syntax}. Syntax transformers are discussed in more depth in Section 6.10.2 [Syntax Rules], page 262, and Section 6.10.3 [Syntax Case], page 267.

\textsuperscript{5} These days such embedded languages are often referred to as \textit{embedded domain-specific languages}, or EDSLs.
**define-syntax** *keyword transformer*  
Bind *keyword* to the syntax transformer obtained by evaluating *transformer*.

After a macro has been defined, further instances of *keyword* in Scheme source code will invoke the syntax transformer defined by *transformer*.

One can also establish local syntactic bindings with **let-syntax**.

**let-syntax** ((*keyword transformer*) . . .) exp1 exp2 . . .  
Bind each *keyword* to its corresponding *transformer* while expanding exp1 exp2 . . . .

A **let-syntax** binding only exists at expansion-time.

\[
\text{(let-syntax ((unless}
  
  
  
  
  
  (unless #t
    
    
    
    
    
  )
)→ "rock rock rock"

A **define-syntax** form is valid anywhere a definition may appear: at the top-level, or locally. Just as a local **define** expands out to an instance of **letrec**, a local **define-syntax** expands out to **letrec-syntax**.

**letrec-syntax** ((*keyword transformer*) . . .) exp1 exp2 . . .  
Bind each *keyword* to its corresponding *transformer* while expanding exp1 exp2 . . . .

In the spirit of **letrec** versus **let**, an expansion produced by *transformer* may reference a *keyword* bound by the same **letrec-syntax**.

\[
\text{(letrec-syntax ((my-or}
  
  
  
  
  
  (my-or #f "rockaway beach")
)→ "rockaway beach"

### 6.10.2 Syntax-rules Macros

**syntax-rules** macros are simple, pattern-driven syntax transformers, with a beauty worthy of Scheme.
Create a syntax transformer that will rewrite an expression using the rules embodied
in the pattern and template clauses.

A syntax-rules macro consists of three parts: the literals (if any), the patterns, and
as many templates as there are patterns.

When the syntax expander sees the invocation of a syntax-rules macro, it matches the
expression against the patterns, in order, and rewrites the expression using the template
from the first matching pattern. If no pattern matches, a syntax error is signalled.

### 6.10.2.1 Patterns

We have already seen some examples of patterns in the previous section: (unless
condition exp ...), (my-or exp), and so on. A pattern is structured like the expression
that it is to match. It can have nested structure as well, like (let ((var val) ...)
exps exp* ...). Broadly speaking, patterns are made of lists, improper lists, vectors,
identifiers, and datums. Users can match a sequence of patterns using the ellipsis (...).

Identifiers in a pattern are called literals if they are present in the syntax-rules
literals list, and pattern variables otherwise. When building up the macro output, the expander
replaces instances of a pattern variable in the template with the matched subexpression.

```scheme
(define-syntax kwote
  (syntax-rules ()
    ((kwote exp)
      (quote exp))))
(kwote (foo . bar))
⇒ (foo . bar)
```

An improper list of patterns matches as rest arguments do:

```scheme
(define-syntax let1
  (syntax-rules ()
    (_ (var val) . exps)
    (let ((var val)) . exps)))
```

However this definition of let1 probably isn’t what you want, as the tail pattern exps
will match non-lists, like (let1 (foo ’bar) . baz). So often instead of using improper lists
as patterns, ellipsized patterns are better. Instances of a pattern variable in the template
must be followed by an ellipsis.

```scheme
(define-syntax let1
  (syntax-rules ()
    (_ (var val) exp exp* ...)
    (let ((var val)) exp exp* ...)))
```

This let1 probably still doesn’t do what we want, because the body matches sequences
of zero expressions, like (let1 (foo ’bar)). In this case we need to assert we have at least
one body expression. A common idiom for this is to name the ellipsized pattern variable
with an asterisk:

```scheme
(define-syntax let1
  (syntax-rules ()
    (_ (var val) exp exp* ...)
    (let ((var val)) exp exp* ...)))
```
(let ((var val)) exp exp* ...)))

A vector of patterns matches a vector whose contents match the patterns, including
ellipsizing and tail patterns.

(define-syntax letv
(syntax-rules ()
  ((_ #((var val) ...) exp exp* ...)  
    (let ((var val) ...) exp exp* ...)))
(letv #((foo 'bar)) foo)
⇒ bar

Literals are used to match specific datums in an expression, like the use of => and else in cond expressions.

(define-syntax cond1
(syntax-rules (=> else)
  ((cond1 test => fun)  
    (let ((exp test))  
      (if exp (fun exp) #f)))
  ((cond1 test exp exp* ...)  
    (if test (begin exp exp* ...)))
  ((cond1 else exp exp* ...)  
    (begin exp exp* ...))))

(define (square x) (* x x))
(cond1 10 => square)
⇒ 100
(let ((=> #t))
  (cond1 10 => square))
⇒ #<procedure square (x)>

A literal matches an input expression if the input expression is an identifier with the
same name as the literal, and both are unbound.

If a pattern is not a list, vector, or an identifier, it matches as a literal, with equal?.

(define-syntax define-matcher-macro
(syntax-rules ()
  ((_ name lit)
    (define-syntax name
      (syntax-rules ()
        ((_ lit) #t)
        ((_ else) #f)))))

(define-matcher-macro is-literal-foo? "foo")
(is-literal-foo? "foo")
⇒ #t
(is-literal-foo? "bar")

Language lawyers probably see the need here for use of literal-identifier=? rather than
free-identifier=?, and would probably be correct. Patches accepted.
The last example indicates that matching happens at expansion-time, not at run-time.

Syntax-rules macros are always used as `(macro . args)`, and the `macro` will always be a symbol. Correspondingly, a `syntax-rules` pattern must be a list (proper or improper), and the first pattern in that list must be an identifier. Incidentally it can be any identifier – it doesn’t have to actually be the name of the macro. Thus the following three are equivalent:

```lisp
(define-syntax when
  (syntax-rules ()
    ((when c e ...)
      (if c (begin e ...))))

(define-syntax when
  (syntax-rules ()
    ((_ c e ...)
      (if c (begin e ...))))

(define-syntax when
  (syntax-rules ()
    ((something-else-entirely c e ...)
      (if c (begin e ...)))
```

For clarity, use one of the first two variants. Also note that since the pattern variable will always match the macro itself (e.g., `cond1`), it is actually left unbound in the template.

### 6.10.2.2 Hygiene

`syntax-rules` macros have a magical property: they preserve referential transparency. When you read a macro definition, any free bindings in that macro are resolved relative to the macro definition; and when you read a macro instantiation, all free bindings in that expression are resolved relative to the expression.

This property is sometimes known as hygiene, and it does aid in code cleanliness. In your macro definitions, you can feel free to introduce temporary variables, without worrying about inadvertently introducing bindings into the macro expansion.

Consider the definition of `my-or` from the previous section:

```lisp
(define-syntax my-or
  (syntax-rules ()
    ((my-or)
      #t)
    ((my-or exp)
      exp)
    ((my-or exp rest ...)
      (let ((t exp))
        (if t
            t
            ...)
A naive expansion of `(let ((t #t)) (my-or #f t))` would yield:

```
(let ((t #t))
 (let ((t #f))
  (if t t t))
⇒ #f
```

Which clearly is not what we want. Somehow the `t` in the definition is distinct from the `t` at the site of use; and it is indeed this distinction that is maintained by the syntax expander, when expanding hygienic macros.

This discussion is mostly relevant in the context of traditional Lisp macros (see Section 6.10.5 [Defmacros], page 275), which do not preserve referential transparency. Hygiene adds to the expressive power of Scheme.

### 6.10.2.3 Shorthands

One often ends up writing simple one-clause `syntax-rules` macros. There is a convenient shorthand for this idiom, in the form of `define-syntax-rule`.

```
define-syntax-rule (keyword . pattern) [docstring] template
                     [Syntax]
```

Define `keyword` as a new `syntax-rules` macro with one clause.

Cast into this form, our `when` example is significantly shorter:

```
(define-syntax-rule (when c e ...) (if c (begin e ...))
```

### 6.10.2.4 Reporting Syntax Errors in Macros

```
syntax-error message [arg ...] [Syntax]
```

Report an error at macro-expansion time. `message` must be a string literal, and the optional `arg` operands can be arbitrary expressions providing additional information.

`syntax-error` is intended to be used within `syntax-rules` templates. For example:

```
(define-syntax simple-let
 (syntax-rules ()
  ((_ (head ... ((x . y) val) . tail) body1 body2 ...) 
   (syntax-error
    "expected an identifier but got" 
    (x . y)))
  ((_ ((name val) ...) body1 body2 ...) 
   ((lambda (name ...) body1 body2 ...) 
    val ...))))
```

### 6.10.2.5 Specifying a Custom Ellipsis Identifier

When writing macros that generate macro definitions, it is convenient to use a different ellipsis identifier at each level. Guile allows the desired ellipsis identifier to be specified as the first operand to `syntax-rules`, as specified by SRFI-46 and R7RS. For example:

```
(define-syntax define-quotation-macros
```

```
(syntax-rules ()
  ((_ (macro-name head-symbol) ...) (begin (define-syntax macro-name
    (syntax-rules ::: ()
      ((_ x :::) (quote (head-symbol x :::))))
    ...)))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)

### 6.10.2.6 Further Information

For a formal definition of `syntax-rules` and its pattern language, see See Section “Macros” in Revised(5) Report on the Algorithmic Language Scheme.

`syntax-rules` macros are simple and clean, but do they have limitations. They do not lend themselves to expressive error messages: patterns either match or they don’t. Their ability to generate code is limited to template-driven expansion; often one needs to define a number of helper macros to get real work done. Sometimes one wants to introduce a binding into the lexical context of the generated code; this is impossible with `syntax-rules`. Relatedly, they cannot programmatically generate identifiers.

The solution to all of these problems is to use `syntax-case` if you need its features. But if for some reason you’re stuck with `syntax-rules`, you might enjoy Joe Marshall’s `syntax-rules` Primer for the Merely Eccentric (http://sites.google.com/site/evalapply/eccentric.txt).

### 6.10.3 Support for the `syntax-case` System

`syntax-case` macros are procedural syntax transformers, with a power worthy of Scheme.

`syntax-case` syntax literals (pattern [guard] exp) . . . [Syntax]

Match the syntax object `syntax` against the given patterns, in order. If a pattern matches, return the result of evaluating the associated `exp`.

Compare the following definitions of `when`:

```scheme
(define-syntax when
  (syntax-rules ()
    ((_ test e e* ...) (if test (begin e e* ...))))))
```

```scheme
(define-syntax when
  (lambda (x)
    (syntax-case x ()
      ((_ test e e* ...)="#'(if test (begin e e* ...))))))
```

Clearly, the `syntax-case` definition is similar to its `syntax-rules` counterpart, and equally clearly there are some differences. The `syntax-case` definition is wrapped in a `lambda`, a function of one argument; that argument is passed to the `syntax-case` invocation; and the “return value” of the macro has a `#'` prefix.
All of these differences stem from the fact that `syntax-case` does not define a syntax transformer itself – instead, `syntax-case` expressions provide a way to destructure a syntax object, and to rebuild syntax objects as output.

So the `lambda` wrapper is simply a leaky implementation detail, that syntax transformers are just functions that transform syntax to syntax. This should not be surprising, given that we have already described macros as “programs that write programs”. `syntax-case` is simply a way to take apart and put together program text, and to be a valid syntax transformer it needs to be wrapped in a procedure.

Unlike traditional Lisp macros (see Section 6.10.5 [Defmacros], page 275), `syntax-case` macros transform syntax objects, not raw Scheme forms. Recall the naive expansion of `my-or` given in the previous section:

```scheme
(let ((t #t))
  (my-or #f t))
;; naive expansion:
(let ((t #t))
  (let ((t #f))
    (if t t t)))
```

Raw Scheme forms simply don’t have enough information to distinguish the first two `t` instances in `(if t t t)` from the third `t`. So instead of representing identifiers as symbols, the syntax expander represents identifiers as annotated syntax objects, attaching such information to those syntax objects as is needed to maintain referential transparency.

**Syntax form**

Create a syntax object wrapping `form` within the current lexical context.

Syntax objects are typically created internally to the process of expansion, but it is possible to create them outside of syntax expansion:

```scheme
(syntax (foo bar baz))
⇒ #<some representation of that syntax>
```

However it is more common, and useful, to create syntax objects when building output from a `syntax-case` expression.

```scheme
(define-syntax add1
  (lambda (x)
    (syntax-case x ()
      ((_ exp)
       (syntax (+ exp 1)))))
```

It is not strictly necessary for a `syntax-case` expression to return a syntax object, because `syntax-case` expressions can be used in helper functions, or otherwise used outside of syntax expansion itself. However a syntax transformer procedure must return a syntax object, so most uses of `syntax-case` do end up returning syntax objects.

Here in this case, the form that built the return value was `(+ exp 1)`. The interesting thing about this is that within a `syntax` expression, any appearance of a pattern variable is substituted into the resulting syntax object, carrying with it all relevant metadata from the source expression, such as lexical identity and source location.
Indeed, a pattern variable may only be referenced from inside a syntax form. The syntax expander would raise an error when defining add1 if it found exp referenced outside a syntax form.

Since syntax appears frequently in macro-heavy code, it has a special reader macro: #'. #’foo is transformed by the reader into (syntax foo), just as ‘foo is transformed into (quote foo).

The pattern language used by syntax-case is conveniently the same language used by syntax-rules. Given this, Guile actually defines syntax-rules in terms of syntax-case:

\[
\begin{align*}
\text{(define-syntax syntax-rules} & \quad \text{(lambda (x)} \\
& \quad \quad \text{(syntax-case x ()} \\
& \quad \quad \quad \text{((_, (k ...) ((keyword . pattern) template) ...)} \\
& \quad \quad \quad \quad \text{#'(lambda (x)} \\
& \quad \quad \quad \quad \text{(syntax-case x (k ...)} \\
& \quad \quad \quad \quad \quad \text{((dummy . pattern) #'template)} \\
& \quad \quad \quad \quad \quad \quad \text{...)))))})
\end{align*}
\]

And that’s that.

6.10.3.1 Why syntax-case?

The examples we have shown thus far could just as well have been expressed with syntax-rules, and have just shown that syntax-case is more verbose, which is true. But there is a difference: syntax-case creates procedural macros, giving the full power of Scheme to the macro expander. This has many practical applications.

A common desire is to be able to match a form only if it is an identifier. This is impossible with syntax-rules, given the datum matching forms. But with syntax-case it is easy:

\[
\begin{align*}
\text{identifier? syntax-object} & \quad \text{[Scheme Procedure]} \\
\text{Returns #t if syntax-object is an identifier, or #f otherwise.}
\end{align*}
\]

;; relying on previous add1 definition
(define-syntax add1!
  (lambda (x)
    (syntax-case x ()
      (((_ var) (identifier? #'var) 
        #'(set! var (add1 var))))))

(define foo 0)
(add1! foo)
foo ⇒ 1
(add1! "not-an-identifier") ⇒ error

With syntax-rules, the error for (add1! "not-an-identifier") would be something like “invalid set!”. With syntax-case, it will say something like “invalid add1!”, because we attach the guard clause to the pattern: (identifier? #'var). This becomes more important with more complicated macros. It is necessary to use identifier?, because to the expander, an identifier is more than a bare symbol.

Note that even in the guard clause, we reference the var pattern variable within a syntax form, via #'var.
Another common desire is to introduce bindings into the lexical context of the output expression. One example would be in the so-called “anaphoric macros”, like aif. Anaphoric macros bind some expression to a well-known identifier, often it, within their bodies. For example, in (aif (foo) (bar it)), it would be bound to the result of (foo).

To begin with, we should mention a solution that doesn’t work:

```scheme
;; doesn’t work
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
        #'(let ((it test))
            (if it then else))))))
```

The reason that this doesn’t work is that, by default, the expander will preserve referential transparency; the then and else expressions won’t have access to the binding of it.

But they can, if we explicitly introduce a binding via `datum->syntax`.

```
[Scheme Procedure]
datum->syntax template-id datum
Create a syntax object that wraps datum, within the lexical context corresponding to the identifier template-id.
```

For completeness, we should mention that it is possible to strip the metadata from a syntax object, returning a raw Scheme datum:

```
[Scheme Procedure]
syntax->datum syntax-object
Strip the metadata from syntax-object, returning its contents as a raw Scheme datum.
```

In this case we want to introduce it in the context of the whole expression, so we can create a syntax object as `(datum->syntax x ‘it)`, where x is the whole expression, as passed to the transformer procedure.

Here’s another solution that doesn’t work:

```scheme
;; doesn’t work either
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
        (let ((it (datum->syntax x ‘it)))
          #'(let ((it test))
              (if it then else)))))))
```

The reason that this one doesn’t work is that there are really two environments at work here – the environment of pattern variables, as bound by `syntax-case`, and the environment of lexical variables, as bound by normal Scheme. The outer let form establishes a binding in the environment of lexical variables, but the inner let form is inside a syntax form, where only pattern variables will be substituted. Here we need to introduce a piece of the lexical environment into the pattern variable environment, and we can do so using `syntax-case` itself:

```scheme
;; works, but is obtuse
```
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       ;; invoking syntax-case on the generated
       ;; syntax object to expose it to ‘syntax’
      (syntax-case (datum->syntax x 'it) ()
        (it
          #'(let ((it test))
              (if it then else))))))
)

(aif (getuid) (display it) (display "none")) (newline)
⊣ 500

However there are easier ways to write this. with-syntax is often convenient:

with-syntax ((pat val) . . .) exp . . .
  [Syntax]
  Bind patterns pat from their corresponding values val, within the lexical context of exp . . .

    ;; better
    (define-syntax aif
      (lambda (x)
        (syntax-case x ()
          ((_ test then else)
            (with-syntax ((it (datum->syntax x 'it)))
              #'(let ((it test))
                  (if it then else))))))

As you might imagine, with-syntax is defined in terms of syntax-case. But even that
might be off-putting to you if you are an old Lisp macro hacker, used to building macro
output with quasiquote. The issue is that with-syntax creates a separation between the
point of definition of a value and its point of substitution.

So for cases in which a quasiquote style makes more sense, syntax-case also de-
defines Quasisyntax, and the related unsyntax and unsyntax-splicing, abbreviated by
the reader as #’, #, and #,0, respectively.

For example, to define a macro that inserts a compile-time timestamp into a source file,
one may write:

(define-syntax display-compile-timestamp
  (lambda (x)
    (syntax-case x ()
      (_)
      #'(begin
          (display "The compile timestamp was: ")
          (display #,(current-time))
          (newline))))))

Readers interested in further information on syntax-case macros should see R. Kent
Dybvig’s excellent The Scheme Programming Language, either edition 3 or 4, in the chapter
on syntax. Dybvig was the primary author of the syntax-case system. The book itself is available online at http://scheme.com/tspl4/.

### 6.10.3.2 Custom Ellipsis Identifiers for syntax-case Macros

When writing procedural macros that generate macro definitions, it is convenient to use a different ellipsis identifier at each level. Guile supports this for procedural macros using the with-ellipsis special form:

```scheme
(with-ellipsis ellipsis body ...)
```

ellipsis must be an identifier. Evaluate body in a special lexical environment such that all macro patterns and templates within body will use ellipsis as the ellipsis identifier instead of the usual three dots (...).

For example:

```scheme
(define-syntax define-quotation-macros
  (lambda (x)
    (syntax-case x ()
      ((_ (macro-name head-symbol) ...) #'(begin (define-syntax macro-name
          (lambda (x)
            (with-ellipsis :::
              (syntax-case x ()
                ((_ x ::::)
                  #'(quote (head-symbol x ::::))))))
        ...)))))

(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(define-quotation-macros (quote-a 1 2 3) (quote-b 1 2 3) (quote-c 1 2 3))
```

Note that with-ellipsis does not affect the ellipsis identifier of the generated code, unless with-ellipsis is included around the generated code.

### 6.10.4 Syntax Transformer Helpers

As noted in the previous section, Guile’s syntax expander operates on syntax objects. Procedural macros consume and produce syntax objects. This section describes some of the auxiliary helpers that procedural macros can use to compare, generate, and query objects of this data type.

#### bound-identifier=? a b

[Scheme Procedure]

Return #t if the syntax objects a and b refer to the same lexically-bound identifier, or #f otherwise.

#### free-identifier=? a b

[Scheme Procedure]

Return #t if the syntax objects a and b refer to the same free identifier, or #f otherwise.

#### generate-temporaries ls

[Scheme Procedure]

Return a list of temporary identifiers as long as ls is long.
And now, a bit of confession time. Guile’s syntax expander originates in code from Chez Scheme: a version of the expander in Chez Scheme that was made portable to other Scheme systems. Way back in the mid-1990s, some Scheme systems didn’t even have the ability to define new abstract data types. For this reason, the portable expander from Chez Scheme that Guile inherited used tagged vectors as syntax objects: vectors whose first element was the symbol, syntax-object.

At the time of this writing it is 2017 and Guile still has support for this strategy. It worked for this long because no one ever puts a literal vector in the operator position:

```
(#(syntax-object ...) 1 2 3)
```

But this state of affairs was an error. Because syntax objects are just vectors, this makes it possible for any Scheme code to forge a syntax object which might cause it to violate abstraction boundaries. You can’t build a sandboxing facility that limits the set of bindings in scope when one can always escape that limit just by evaluating a special vector. To fix this problem, Guile 2.2.1 finally migrated to represent syntax objects as a distinct type with a distinct constructor that is unavailable to user code.

However, Guile still has to support “legacy” syntax objects, because it could be that a file compiled with Guile 2.2.0 embeds syntax objects of the vector kind. Whether the expander treats the special tagged vectors as syntax objects is now controllable by the allow-legacy-syntax-objects? parameter:

```
allow-legacy-syntax-objects? [Scheme Procedure]
  A parameter that indicates whether the expander should support legacy syntax objects, as described above. For ABI stability reasons, the default is #t. Use parameterize to bind it to #f. See Section 6.13.12 [Parameters], page 324.
```

Guile also offers some more experimental interfaces in a separate module. As was the case with the Large Hadron Collider, it is unclear to our senior macrologists whether adding these interfaces will result in awesomeness or in the destruction of Guile via the creation of a singularity. We will preserve their functionality through the 2.0 series, but we reserve the right to modify them in a future stable series, to a more than usual degree.

```
(use-modules (system syntax))
```

```
syntax-module id [Scheme Procedure]
Return the name of the module whose source contains the identifier id.
```

```
syntax-local-binding id [#:resolve-syntax-parameters?=#$t] [Scheme Procedure]
  Resolve the identifier id, a syntax object, within the current lexical environment, and return two values, the binding type and a binding value. The binding type is a symbol, which may be one of the following:

  lexical A lexically-bound variable. The value is a unique token (in the sense of eq?) identifying this binding.
```
macro  A syntax transformer, either local or global. The value is the transformer procedure.

syntax-parameter  
A syntax parameter (see Section 6.10.7 [Syntax Parameters], page 278). By default, syntax-local-binding will resolve syntax parameters, so that this value will not be returned. Pass #:resolve-syntax-parameters? #f to indicate that you are interested in syntax parameters. The value is the default transformer procedure, as in macro.

pattern-variable  
A pattern variable, bound via syntax-case. The value is an opaque object, internal to the expander.

ellipsis  
An internal binding, bound via with-ellipsis. The value is the (anti-marked) local ellipsis identifier.

displaced-lexical  
A lexical variable that has gone out of scope. This can happen if a badly-written procedural macro saves a syntax object, then attempts to introduce it in a context in which it is unbound. The value is #f.

global  
A global binding. The value is a pair, whose head is the symbol, and whose tail is the name of the module in which to resolve the symbol.

other  
Some other binding, like lambda or other core bindings. The value is #f.

This is a very low-level procedure, with limited uses. One case in which it is useful is to build abstractions that associate auxiliary information with macros:

```
(define aux-property (make-object-property))
(define-syntax-rule (with-aux aux value)
  (let ((trans value))
    (set! (aux-property trans) aux)
    trans))
(define-syntax retrieve-aux
  (lambda (x)
    (syntax-case x ()
      ((x id)
       (call-with-values (lambda () (syntax-local-binding #'id))
        (lambda (type val)
          (with-syntax ((aux (datum->syntax #'here
                                      (and (eq? type 'macro)
                                          (aux-property val)))))
            #''aux)))))
  (define-syntax foo
    (with-aux 'bar
      (syntax-rules () ((_) 'foo))))
  (foo)
⇒ foo
(retrieve-aux foo)
⇒ bar
```
**syntax-local-binding** must be called within the dynamic extent of a syntax transformer; to call it otherwise will signal an error.

**syntax-locally-bound-identifiers** id  
[Scheme Procedure]  
Return a list of identifiers that were visible lexically when the identifier id was created, in order from outermost to innermost.

This procedure is intended to be used in specialized procedural macros, to provide a macro with the set of bound identifiers that the macro can reference.

As a technical implementation detail, the identifiers returned by **syntax-locally-bound-identifiers** will be anti-marked, like the syntax object that is given as input to a macro. This is to signal to the macro expander that these bindings were present in the original source, and do not need to be hygienically renamed, as would be the case with other introduced identifiers. See the discussion of hygiene in section 12.1 of the R6RS, for more information on marks.

```scheme
(define (local-lexicals id)
  (filter (lambda (x)
             (eq? (syntax-local-binding x) 'lexical))
             (syntax-locally-bound-identifiers id)))

(define-syntax lexicals
  (lambda (x)
    (syntax-case x ()
      ((lexicals) #'(lexicals lexicals))
      ((lexicals scope)
        (with-syntax (((id ...) (local-lexicals #'scope))
                      '#(list (cons 'id id) ...)))
        #'(list (cons 'id id) ...))))

(let* ((x 10) (x 20)) (lexicals))
⇒ ((x . 10) (x . 20))
```

### 6.10.5 Lisp-style Macro Definitions

The traditional way to define macros in Lisp is very similar to procedure definitions. The key differences are that the macro definition body should return a list that describes the transformed expression, and that the definition is marked as a macro definition (rather than a procedure definition) by the use of a different definition keyword: in Lisp, **defmacro** rather than **defun**, and in Scheme, **define-macro** rather than **define**.

Guile supports this style of macro definition using both **defmacro** and **define-macro**. The only difference between them is how the macro name and arguments are grouped together in the definition:

```
(defmacro name (args ...) body ...)
```

is the same as

```
(define-macro (name args ...) body ...)
```

The difference is analogous to the corresponding difference between Lisp’s **defun** and Scheme’s **define**.

Having read the previous section on **syntax-case**, it’s probably clear that Guile actually implements defmacros in terms of **syntax-case**, applying the transformer on the expression
between invocations of `syntax->datum` and `datum->syntax`. This realization leads us to
the problem with defmacros, that they do not preserve referential transparency. One can be
careful to not introduce bindings into expanded code, via liberal use of `gensym`, but there is
no getting around the lack of referential transparency for free bindings in the macro itself.

Even a macro as simple as our `when` from before is difficult to get right:

```
(define-macro (when cond exp . rest)
  '(if ,cond
    (begin ,exp . ,rest)))
```

```
(when #f (display "Launching missiles!\n"))
⇒ #f
```

```
(let ((if list))
  (when #f (display "Launching missiles!\n")))
⇒ ( #f #<unspecified>)
```

Guile’s perspective is that defmacros have had a good run, but that modern macros
should be written with `syntax-rules` or `syntax-case`. There are still many uses of def-
macros within Guile itself, but we will be phasing them out over time. Of course we won’t
take away `defmacro` or `define-macro` themselves, as there is lots of code out there that
uses them.

### 6.10.6 Identifier Macros

When the syntax expander sees a form in which the first element is a macro, the whole
form gets passed to the macro’s syntax transformer. One may visualize this as:

```
(define-syntax foo foo-transformer)
(foo arg...)
;; expands via
(foo-transformer '#(foo arg...))
```

If, on the other hand, a macro is referenced in some other part of a form, the syntax
transformer is invoked with only the macro reference, not the whole form.

```
(define-syntax foo foo-transformer)
foo
;; expands via
(foo-transformer '#foo)
```

This allows bare identifier references to be replaced programmatically via a macro.
`syntax-rules` provides some syntax to effect this transformation more easily.

```
(identifier-syntax exp)  [Syntax]
```

Returns a macro transformer that will replace occurrences of the macro with `exp`.

For example, if you are importing external code written in terms of `fx+`, the fixnum
addition operator, but Guile doesn’t have `fx+`, you may use the following to replace `fx+
with `+`:

```
(define-syntax fx+ (identifier-syntax +))
```
There is also special support for recognizing identifiers on the left-hand side of a `set!` expression, as in the following:

```
(define-syntax foo foo-transformer)
(set! foo val)
;; expands via
(foo-transformer #'(set! foo val))
;; if foo-transformer is a "variable transformer"
```

As the example notes, the transformer procedure must be explicitly marked as being a “variable transformer”, as most macros aren’t written to discriminate on the form in the operator position.

**make-variable-transformer** transformer

[Scheme Procedure]

Mark the `transformer` procedure as being a “variable transformer”. In practice this means that, when bound to a syntactic keyword, it may detect references to that keyword on the left-hand-side of a `set!`.

```
(define bar 10)
(define-syntax bar-alias
  (make-variable-transformer
    (lambda (x)
      (syntax-case x (set!)
        ((set! var val) #'(set! bar val))
        ((var arg ...) #'(bar arg ...))
        (var (identifier? #'var) #'bar)))))

bar-alias ⇒ 10
(set! bar-alias 20)
bar ⇒ 20
(set! bar 30)
bar-alias ⇒ 30
```

There is an extension to identifier-syntax which allows it to handle the `set!` case as well:

**identifier-syntax** (var exp1) ((set! var val) exp2)

[Syntax]

Create a variable transformer. The first clause is used for references to the variable in operator or operand position, and the second for appearances of the variable on the left-hand-side of an assignment.

For example, the previous `bar-alias` example could be expressed more succinctly like this:

```
(define-syntax bar-alias
  (identifier-syntax
    (var bar)
    ((set! var val) (set! bar val)))))
```

As before, the templates in `identifier-syntax` forms do not need wrapping in `#` syntax forms.
6.10.7 Syntax Parameters

Syntax parameters\(^7\) are a mechanism for rebinding a macro definition within the dynamic extent of a macro expansion. This provides a convenient solution to one of the most common types of unhygienic macro: those that introduce a unhygienic binding each time the macro is used. Examples include a `lambda` form with a `return` keyword, or class macros that introduce a special `self` binding.

With syntax parameters, instead of introducing the binding unhygienically each time, we instead create one binding for the keyword, which we can then adjust later when we want the keyword to have a different meaning. As no new bindings are introduced, hygiene is preserved. This is similar to the dynamic binding mechanisms we have at run-time (see Section 7.5.27 [SRFI-39], page 624), except that the dynamic binding only occurs during macro expansion. The code after macro expansion remains lexically scoped.

\[\text{define-syntax-parameter} \ \text{keyword} \ \text{transformer} \ \ \ \ [\text{Syntax}]\]

Binds `keyword` to the value obtained by evaluating `transformer`. The `transformer` provides the default expansion for the syntax parameter, and in the absence of `syntax-parameterize`, is functionally equivalent to `define-syntax`. Usually, you will just want to have the `transformer` throw a syntax error indicating that the keyword is supposed to be used in conjunction with another macro, for example:

```scheme
(define-syntax-parameter return
  (lambda (stx)
    (syntax-violation 'return "return used outside of a lambda\^" stx)))
```

\[\text{syntax-parameterize} \ ((\text{keyword} \ \text{transformer}) \ldots) \ \text{exp} \ldots \ \ [\text{Syntax}]\]

Adjusts `keyword` \ldots to use the values obtained by evaluating their `transformer` \ldots, in the expansion of the `exp` \ldots forms. Each `keyword` must be bound to a syntax-parameter. `syntax-parameterize` differs from `let-syntax`, in that the binding is not shadowed, but adjusted, and so uses of the keyword in the expansion of `exp` \ldots use the new transformers. This is somewhat similar to how `parameterize` adjusts the values of regular parameters, rather than creating new bindings.

```scheme
(define-syntax lambda^
  (syntax-rules ()
    [(lambda^ argument-list body body* ...) (lambda argument-list
      (call-with-current-continuation
        (lambda (escape)
          ;; In the body we adjust the 'return' keyword so that calls
          ;; to 'return' are replaced with calls to the escape
          ;; continuation.
          (syntax-parameterize ([return (syntax-rules ()
                                       [(return vals (... ...))
                                         (escape vals (... ...))]]))
            body body* ...))))])))
```

```
;; Now we can write functions that return early. Here, 'product' will
```

\(^{7}\) Described in the paper *Keeping it Clean with Syntax Parameters* by Barzilay, Culpepper and Flatt.
(define product
  (lambda^ (list)
    (fold (lambda (n o)
       (if (zero? n)
           (return 0)
           (* n o)))
       1
       list)))

6.10.8 Eval-when

As syntax-case macros have the whole power of Scheme available to them, they present a problem regarding time: when a macro runs, what parts of the program are available for the macro to use?

The default answer to this question is that when you import a module (via define-module or use-modules), that module will be loaded up at expansion-time, as well as at run-time. Additionally, top-level syntactic definitions within one compilation unit made by define-syntax are also evaluated at expansion time, in the order that they appear in the compilation unit (file).

But if a syntactic definition needs to call out to a normal procedure at expansion-time, it might well need need special declarations to indicate that the procedure should be made available at expansion-time.

For example, the following code will work at a REPL, but not in a file:

;; incorrect
(use-modules (srfi srfi-19))
(define (date) (date->string (current-date)))
(define-syntax %date (identifier-syntax (date)))
(define *compilation-date* %date)

It works at a REPL because the expressions are evaluated one-by-one, in order, but if placed in a file, the expressions are expanded one-by-one, but not evaluated until the compiled file is loaded.

The fix is to use eval-when.

;; correct: using eval-when
(use-modules (srfi srfi-19))
(eval-when (expand load eval)
  (define (date) (date->string (current-date)))
  (define-syntax %date (identifier-syntax (date)))
  (define *compilation-date* %date)

```
eval-when conditions exp... [Syntax]
Evaluate exp... under the given conditions. Valid conditions include:

expand       Evaluate during macro expansion, whether compiling or not.
load         Evaluate during the evaluation phase of compiled code, e.g. when loading a compiled module or running compiled code at the REPL.
```
eval Evaluate during the evaluation phase of non-compiled code.

compile Evaluate during macro expansion, but only when compiling.

In other words, when using the primitive evaluator, eval-when expressions with expand are run during macro expansion, and those with eval are run during the evaluation phase.

When using the compiler, eval-when expressions with either expand or compile are run during macro expansion, and those with load are run during the evaluation phase.

When in doubt, use the three conditions (expand load eval), as in the example above. Other uses of eval-when may void your warranty or poison your cat.

### 6.10.9 Macro Expansion

Usually, macros are expanded on behalf of the user as needed. Macro expansion is an integral part of eval and compile. Users can also expand macros at the REPL prompt via the expand REPL command; See Section 4.4.4.4 [Compile Commands], page 50.

Macros can also be expanded programmatically, via macroexpand, but the details get a bit hairy for two reasons.

The first complication is that the result of macro-expansion isn’t Scheme: it’s Tree-IL, Guile’s high-level intermediate language. See Section 9.4.3 [Tree-IL], page 839. As “hygienic macros” can produce identifiers that are distinct but have the same name, the output format needs to be able to represent distinctions between variable identities and names. Again, See Section 9.4.3 [Tree-IL], page 839, for all the details. The easiest thing is to just run tree-il->scheme on the result of macro-expansion:

```
(macroexpand '(+ 1 2))
⇒
#<tree-il (call (toplevel +) (const 1) (const 2))>
```

```
(use-modules (language tree-il))
(tree-il->scheme (macroexpand '(+ 1 2)))
⇒
(+ 1 2)
```

The second complication involves eval-when. As an example, what would it mean to macro-expand the definition of a macro?

```
(macroexpand '(define-syntax qux (identifier-syntax 'bar)))
⇒
?
```

The answer is that it depends who is macro-expanding, and why. Do you define the macro in the current environment? Residualize a macro definition? Both? Neither? The default is to expand in “eval” mode, which means an eval-when clauses will only proceed when eval (or expand) is in its condition set. Top-level macros will be eval’d in the top-level environment.

In this way (macroexpand foo) is equivalent to (macroexpand foo 'e '(eval)). The second argument is the mode ('e for “eval”) and the third is the eval-syntax-expanders-when parameter (only eval in this default setting).
But if you are compiling the macro definition, probably you want to reify the macro
definition itself. In that case you pass 'c as the second argument to \texttt{macroexpand}. But
probably you want the macro definition to be present at compile time as well, so you
pass '(\texttt{compile load eval}) as the esew parameter. In fact (\texttt{compile foo #:\texttt{to 'tree-nil}})
is entirely equivalent to (\texttt{macroexpand foo 'c '(\texttt{compile load eval}))}; See Section 9.4.2
[The Scheme Compiler], page 838.

It’s a terrible interface; we know. The macroexpander is somewhat tricksy regarding
modes, so unless you are building a macro-expanding tool, we suggest to avoid invoking it
directly.

\subsection*{6.10.10 Hygiene and the Top-Level}

Consider the following macro.

\begin{verbatim}
(define-syntax-rule (defconst name val)
    (begin
        (define t val)
        (define-syntax-rule (name) t)))
\end{verbatim}

If we use it to make a couple of bindings:

\begin{verbatim}
(defconst foo 42)
(defconst bar 37)
\end{verbatim}

The expansion would look something like this:

\begin{verbatim}
(begin
    (define t 42)
    (define-syntax-rule (foo) t))
(begin
    (define t 37)
    (define-syntax-rule (bar) t))
\end{verbatim}

As the two t bindings were introduced by the macro, they should be introduced hygien-
ically -- and indeed they are, inside a lexical contour (a \texttt{let} or some other lexical scope).
The t reference in foo is distinct to the reference in bar.

At the top-level things are more complicated. Before Guile 2.2, a use of \texttt{defconst} at
the top-level would not introduce a fresh binding for t. This was consistent with a weasely
interpretation of the Scheme standard, in which all possible bindings may be assumed to
exist, at the top-level, and in which we merely take advantage of toplevel \texttt{define} of an
existing binding being equivalent to \texttt{set!}. But it’s not a good reason.

The solution is to create fresh names for all bindings introduced by macros -- not just
bindings in lexical contours, but also bindings introduced at the top-level.

However, the obvious strategy of just giving random names to introduced toplevel iden-
tifiers poses a problem for separate compilation. Consider without loss of generality a
\texttt{defconst} of foo in module a that introduces the fresh top-level name \texttt{t-1}. If we then
compile a module b that uses foo, there is now a reference to \texttt{t-1} in module b. If module
a is then expanded again, for whatever reason, for example in a simple recompilation, the
introduced t gets a fresh name; say, \texttt{t-2}. Now module b has broken because module a no
longer has a binding for \texttt{t-1}.

If introduced top-level identifiers “escape” a module, in whatever way, they then form
part of the binary interface (ABI) of a module. It is unacceptable from an engineering
point of view to allow the ABI to change randomly. (It also poses practical problems in meeting the recompilation conditions of the Lesser GPL license, for such modules.) For this reason many people prefer to never use identifier-introducing macros at the top-level, instead making those macros receive the names for their introduced identifiers as part of their arguments, or to construct them programmatically and use \texttt{datum->syntax}. But this approach requires omniscience as to the implementation of all macros one might use, and also limits the expressive power of Scheme macros.

There is no perfect solution to this issue. Guile does a terrible thing here. When it goes to introduce a top-level identifier, Guile gives the identifier a pseudo-fresh name: a name that depends on the hash of the source expression in which the name occurs. The result in this case is that the introduced definitions expand as:

\begin{verbatim}
(begin
  (define t-1dc5e42de7c1050c 42)
  (define-syntax-rule (foo) t-1dc5e42de7c1050c))

(begin
  (define t-10cb8ce9fdd6d6e9 37)
  (define-syntax-rule (bar) t-10cb8ce9fdd6d6e9))
\end{verbatim}

However, note that as the hash depends solely on the expression introducing the definition, we also have:

\begin{verbatim}
(defconst baz 42)
⇒ (begin
  (define t-1dc5e42de7c1050c 42)
  (define-syntax-rule (baz) t-1dc5e42de7c1050c))
\end{verbatim}

Note that the introduced binding has the same name! This is because the source expression, \texttt{(define t 42)}, was the same. Probably you will never see an error in this area, but it is important to understand the components of the interface of a module, and that interface may include macro-introduced identifiers.

### 6.10.11 Internal Macros

\texttt{make-syntax-transformer name type binding} \hspace{1cm} \textbf{[Scheme Procedure]}

Construct a syntax transformer object. This is part of Guile’s low-level support for syntax-case.

\texttt{macro? obj} \hspace{1cm} \textbf{[Scheme Procedure]}

\texttt{scm_macro_p (obj)} \hspace{1cm} \textbf{[C Function]}

Return \#t if \textit{obj} is a syntax transformer, or \#f otherwise.

Note that it’s a bit difficult to actually get a macro as a first-class object; simply naming it (like \texttt{case}) will produce a syntax error. But it is possible to get these objects using \texttt{module-ref}:

\begin{verbatim}
  (macro? (module-ref (current-module) 'case))
⇒ \#t
\end{verbatim}

\texttt{macro-type m} \hspace{1cm} \textbf{[Scheme Procedure]}

\texttt{scm_macro_type (m)} \hspace{1cm} \textbf{[C Function]}

Return the type that was given when \textit{m} was constructed, via \texttt{make-syntax-transformer}.
**Scheme Procedure**
macro-name \( m \)

scm_macro_name \((m)\)

Return the name of the macro \( m \).

**Scheme Procedure**
macro-binding \( m \)

scm_macro_binding \((m)\)

Return the binding of the macro \( m \).

**Scheme Procedure**
macro-transformer \( m \)

scm_macro_transformer \((m)\)

Return the transformer of the macro \( m \). This will return a procedure, for which one may ask the docstring. That’s the whole reason this section is documented. Actually a part of the result of macro-binding.

**6.11 General Utility Functions**

This chapter contains information about procedures which are not cleanly tied to a specific data type. Because of their wide range of applications, they are collected in a utility chapter.

**6.11.1 Equality**

There are three kinds of core equality predicates in Scheme, described below. The same kinds of comparisons arise in other functions, like `memq` and friends (see Section 6.6.9.7 [List Searching], page 185).

For all three tests, objects of different types are never equal. So for instance a list and a vector are not `equal?`, even if their contents are the same. Exact and inexact numbers are considered different types too, and are hence not equal even if their values are the same.

- `eq?` tests just for the same object (essentially a pointer comparison). This is fast, and can be used when searching for a particular object, or when working with symbols or keywords (which are always unique objects).

- `eqv?` extends `eq?` to look at the value of numbers and characters. It can for instance be used somewhat like `=` (see Section 6.6.2.8 [Comparison], page 117) but without an error if one operand isn’t a number.

- `equal?` goes further, it looks (recursively) into the contents of lists, vectors, etc. This is good for instance on lists that have been read or calculated in various places and are the same, just not made up of the same pairs. Such lists look the same (when printed), and `equal?` will consider them the same.

---

**Scheme Procedure**
eq? \( x \ y \)

scm_eq_p \((x, y)\)

Return `#t` if \( x \) and \( y \) are the same object, except for numbers and characters. For example,

```scheme
(define x (vector 1 2 3))
(define y (vector 1 2 3))
```

```scheme
(eq? x x) ⇒ #t
(eq? x y) ⇒ #f
```
Numbers and characters are not equal to any other object, but the problem is they’re not necessarily eq? to themselves either. This is even so when the number comes directly from a variable,

\[
\begin{align*}
(\text{let } ((n (+ 2 3)))
\quad \text{(eq? n n))) & \Rightarrow *\text{unspecified*} \\
\end{align*}
\]

Generally eqv? below should be used when comparing numbers or characters. = (see Section 6.6.2.8 [Comparison], page 117) or char=? (see Section 6.6.3 [Characters], page 129) can be used too.

It’s worth noting that end-of-list (), #t, #f, a symbol of a given name, and a keyword of a given name, are unique objects. There’s just one of each, so for instance no matter how () arises in a program, it’s the same object and can be compared with eq?,

\[
\begin{align*}
(\text{define x (cdr } '(123)))
\quad \text{(define y (cdr } '(456)))
\quad \text{(eq? x y)} & \Rightarrow #t \\
(\text{define x (string->symbol } "\text{foo}\text{")})
\quad \text{(eq? x } '\text{foo}) & \Rightarrow #t \\
\end{align*}
\]

\[
\text{int scm_is_eq (SCM x, SCM y)} \quad \text{[C Function]}
\]

Return 1 when \(x\) and \(y\) are equal in the sense of eq?, otherwise return 0.

The == operator should not be used on SCM values, an SCM is a C type which cannot necessarily be compared using == (see Section 6.3 [The SCM Type], page 100).

\[
\begin{align*}
\text{eqv? x y} \quad \text{[Scheme Procedure]} \\
\text{scm_eqv_p (x, y)} \quad \text{[C Function]}
\end{align*}
\]

Return #t if \(x\) and \(y\) are the same object, or for characters and numbers the same value.

On objects except characters and numbers, eqv? is the same as eq? above, it’s true if \(x\) and \(y\) are the same object.

If \(x\) and \(y\) are numbers or characters, eqv? compares their type and value. An exact number is not eqv? to an inexact number (even if their value is the same).

\[
\begin{align*}
(\text{eqv? } 3 (+ 1 2)) & \Rightarrow #t \\
(\text{eqv? } 1 1.0) & \Rightarrow #f \\
\end{align*}
\]

\[
\begin{align*}
\text{equal? x y} \quad \text{[Scheme Procedure]} \\
\text{scm_equal_p (x, y)} \quad \text{[C Function]}
\end{align*}
\]

Return #t if \(x\) and \(y\) are the same type, and their contents or value are equal.

For a pair, string, vector, array or structure, equal? compares the contents, and does so using the same equal? recursively, so a deep structure can be traversed.

\[
\begin{align*}
(\text{equal? (list 1 2 3) (list 1 2 3)}) & \Rightarrow #t \\
(\text{equal? (list 1 2 3) (vector 1 2 3)}) & \Rightarrow #f \\
\end{align*}
\]
For other objects, \texttt{equal?} compares as per \texttt{eqv?} above, which means characters and numbers are compared by type and value (and like \texttt{eqv?}, exact and inexact numbers are not \texttt{equal?}, even if their value is the same).

\begin{verbatim}
(equal? 3 (+ 1 2)) ⇒ #t
(equal? 1 1.0) ⇒ #f
\end{verbatim}

Hash tables are currently only compared as per \texttt{eq?}, so two different tables are not \texttt{equal?}, even if their contents are the same.

\texttt{equal?} does not support circular data structures, it may go into an infinite loop if asked to compare two circular lists or similar.

GOOPS object types (see Chapter 8 [GOOPS], page 759), including foreign object types (see Section 5.5 [Defining New Foreign Object Types], page 74), can have an \texttt{equal?} implementation specialized on two values of the same type. If \texttt{equal?} is called on two GOOPS objects of the same type, \texttt{equal?} will dispatch out to a generic function. This lets an application traverse the contents or control what is considered \texttt{equal?} for two objects of such a type. If there’s no such handler, the default is to just compare as per \texttt{eq?}.

### 6.11.2 Object Properties

It’s often useful to associate a piece of additional information with a Scheme object even though that object does not have a dedicated slot available in which the additional information could be stored. Object properties allow you to do just that.

Guile’s representation of an object property is a procedure-with-setter (see Section 6.9.8 [Procedures with Setters], page 259) that can be used with the generalized form of \texttt{set!} (REFFIXME) to set and retrieve that property for any Scheme object. So, setting a property looks like this:

\begin{verbatim}
(set! (my-property obj1) value-for-obj1)
(set! (my-property obj2) value-for-obj2)
\end{verbatim}

And retrieving values of the same property looks like this:

\begin{verbatim}
(my-property obj1) ⇒ value-for-obj1
(my-property obj2) ⇒ value-for-obj2
\end{verbatim}

To create an object property in the first place, use the \texttt{make-object-property} procedure:

\begin{verbatim}
(define my-property (make-object-property))
\end{verbatim}

\texttt{make-object-property} \hfill [Scheme Procedure]

Create and return an object property. An object property is a procedure-with-setter that can be called in two ways. \texttt{(set! (property obj) val)} sets \texttt{obj}’s \texttt{property} to \texttt{val}. \texttt{(property obj)} returns the current setting of \texttt{obj}’s \texttt{property}.
A single object property created by `make-object-property` can associate distinct property values with all Scheme values that are distinguishable by `eq?` (ruling out numeric values).

Internally, object properties are implemented using a weak key hash table. This means that, as long as a Scheme value with property values is protected from garbage collection, its property values are also protected. When the Scheme value is collected, its entry in the property table is removed and so the (ex-) property values are no longer protected by the table.

Guile also implements a more traditional Lispy interface to properties, in which each object has an list of key-value pairs associated with it. Properties in that list are keyed by symbols. This is a legacy interface; you should use weak hash tables or object properties instead.

```
object-properties obj
scm_object_properties (obj)
  Return obj’s property list.
set-object-properties! obj alist
scm_set_object_properties_x (obj, alist)
  Set obj’s property list to alist.
object-property obj key
scm_object_property (obj, key)
  Return the property of obj with name key.
set-object-property! obj key value
scm_set_object_property_x (obj, key, value)
  In obj’s property list, set the property named key to value.
```

### 6.11.3 Sorting

Sorting is very important in computer programs. Therefore, Guile comes with several sorting procedures built-in. As always, procedures with names ending in `!` are side-effecting, that means that they may modify their parameters in order to produce their results.

The first group of procedures can be used to merge two lists (which must be already sorted on their own) and produce sorted lists containing all elements of the input lists.

```
merge alist blist less
scm_merge (alist, blist, less)
  Merge two already sorted lists into one. Given two lists alist and blist, such that
  (sorted? alist less?) and (sorted? blist less?), return a new list in which the
  elements of alist and blist have been stably interleaved so that (sorted? (merge
  alist blist less?) less?). Note: this does not accept vectors.
merge! alist blist less
scm_merge_x (alist, blist, less)
  Takes two lists alist and blist such that (sorted? alist less?) and (sorted? blist
  less?) and returns a new list in which the elements of alist and blist have been
  stably interleaved so that (sorted? (merge alist blist less?) less?). This is
  the destructive variant of merge Note: this does not accept vectors.
```
The following procedures can operate on sequences which are either vectors or list. According to the given arguments, they return sorted vectors or lists, respectively. The first of the following procedures determines whether a sequence is already sorted, the other sort a given sequence. The variants with names starting with `stable-` are special in that they maintain a special property of the input sequences: If two or more elements are the same according to the comparison predicate, they are left in the same order as they appeared in the input.

```scheme
sorted? items less
```

```
scm_sorted_p (items, less)
```

Return `#t` if `items` is a list or vector such that, for each element `x` and the next element `y` of `items`, `(less y x)` returns `#f`. Otherwise return `#f`.

```scheme
sort items less
```

```
scm_sort (items, less)
```

Sort the sequence `items`, which may be a list or a vector. `less` is used for comparing the sequence elements. This is not a stable sort.

```scheme
sort! items less
```

```
scm_sort_x (items, less)
```

Sort the sequence `items`, which may be a list or a vector. `less` is used for comparing the sequence elements. The sorting is destructive, that means that the input sequence is modified to produce the sorted result. This is not a stable sort.

```scheme
stable-sort items less
```

```
scm_stable_sort (items, less)
```

Sort the sequence `items`, which may be a list or a vector. `less` is used for comparing the sequence elements. This is a stable sort.

```scheme
stable-sort! items less
```

```
scm_stable_sort_x (items, less)
```

Sort the sequence `items`, which may be a list or a vector. `less` is used for comparing the sequence elements. The sorting is destructive, that means that the input sequence is modified to produce the sorted result. This is a stable sort.

The procedures in the last group only accept lists or vectors as input, as their names indicate.

```scheme
sort-list items less
```

```
scm_sort_list (items, less)
```

Sort the list `items`, using `less` for comparing the list elements. This is a stable sort.

```scheme
sort-list! items less
```

```
scm_sort_list_x (items, less)
```

Sort the list `items`, using `less` for comparing the list elements. The sorting is destructive, that means that the input list is modified to produce the sorted result. This is a stable sort.
restricted-vector-sort!  vec less startpos endpos  [Scheme Procedure]
scm_restricted_vector_sort_x (vec, less, startpos, endpos)  [C Function]
Sort the vector vec, using less for comparing the vector elements. startpos (inclusively) and endpos (exclusively) delimit the range of the vector which gets sorted.
The return value is not specified.

6.11.4 Copying Deep Structures
The procedures for copying lists (see Section 6.6.9 [Lists], page 181) only produce a flat copy of the input list, and currently Guile does not even contain procedures for copying vectors. copy-tree can be used for these application, as it does not only copy the spine of a list, but also copies any pairs in the cars of the input lists.

copy-tree obj  [Scheme Procedure]
scm_copy_tree (obj)  [C Function]
Recursively copy the data tree that is bound to obj, and return the new data structure.
copy-tree recurses down the contents of both pairs and vectors (since both cons cells and vector cells may point to arbitrary objects), and stops recursing when it hits any other object.

6.11.5 General String Conversion
When debugging Scheme programs, but also for providing a human-friendly interface, a procedure for converting any Scheme object into string format is very useful. Conversion from/to strings can of course be done with specialized procedures when the data type of the object to convert is known, but with this procedure, it is often more comfortable.

object->string converts an object by using a print procedure for writing to a string port, and then returning the resulting string. Converting an object back from the string is only possible if the object type has a read syntax and the read syntax is preserved by the printing procedure.

object->string obj [printer]  [Scheme Procedure]
scm_object_to_string (obj, printer)  [C Function]
Return a Scheme string obtained by printing obj. Printing function can be specified by the optional second argument printer (default: write).

6.11.6 Hooks
A hook is a list of procedures to be called at well defined points in time. Typically, an application provides a hook h and promises its users that it will call all of the procedures in h at a defined point in the application’s processing. By adding its own procedure to h, an application user can tap into or even influence the progress of the application.

Guile itself provides several such hooks for debugging and customization purposes: these are listed in a subsection below.

When an application first creates a hook, it needs to know how many arguments will be passed to the hook’s procedures when the hook is run. The chosen number of arguments (which may be none) is declared when the hook is created, and all the procedures that are added to that hook must be capable of accepting that number of arguments.

A hook is created using make-hook. A procedure can be added to or removed from a hook using add-hook! or remove-hook!, and all of a hook’s procedures can be removed
together using \texttt{reset-hook}. When an application wants to run a hook, it does so using \texttt{run-hook}.

\subsection*{6.11.6.1 Hook Usage by Example}

Hook usage is shown by some examples in this section. First, we will define a hook of arity 2 — that is, the procedures stored in the hook will have to accept two arguments.

\begin{verbatim}
(define hook (make-hook 2))

hook ⇒ #<hook 2 40286c90>
\end{verbatim}

Now we are ready to add some procedures to the newly created hook with \texttt{add-hook}. In the following example, two procedures are added, which print different messages and do different things with their arguments.

\begin{verbatim}
(add-hook! hook (lambda (x y)
    (display "Foo: ")
    (display (+ x y))
    (newline)))

(add-hook! hook (lambda (x y)
    (display "Bar: ")
    (display (* x y))
    (newline)))
\end{verbatim}

Once the procedures have been added, we can invoke the hook using \texttt{run-hook}.

\begin{verbatim}
(run-hook hook 3 4)
  ⊣ Bar: 12
  ⊣ Foo: 7
\end{verbatim}

Note that the procedures are called in the reverse of the order with which they were added. This is because the default behaviour of \texttt{add-hook} is to add its procedure to the \texttt{front} of the hook’s procedure list. You can force \texttt{add-hook} to add its procedure to the \texttt{end} of the list instead by providing a third \texttt{#t} argument on the second call to \texttt{add-hook}.

\begin{verbatim}
(add-hook! hook (lambda (x y)
    (display "Foo: ")
    (display (+ x y))
    (newline)))

(add-hook! hook (lambda (x y)
    (display "Bar: ")
    (display (* x y))
    (newline)) #t) ; Change here!
\end{verbatim}

\begin{verbatim}
(run-hook hook 3 4)
  ⊣ Foo: 7
  ⊣ Bar: 12
\end{verbatim}

\subsection*{6.11.6.2 Hook Reference}

When you create a hook with \texttt{make-hook}, you must specify the arity of the procedures which can be added to the hook. If the arity is not given explicitly as an argument to \texttt{make-hook},
it defaults to zero. All procedures of a given hook must have the same arity, and when the procedures are invoked using \texttt{run-hook}, the number of arguments passed must match the arity specified at hook creation time.

The order in which procedures are added to a hook matters. If the third parameter to \texttt{add-hook!} is omitted or is equal to \texttt{#f}, the procedure is added in front of the procedures which might already be on that hook, otherwise the procedure is added at the end. The procedures are always called from the front to the end of the list when they are invoked via \texttt{run-hook}.

The ordering of the list of procedures returned by \texttt{hook->list} matches the order in which those procedures would be called if the hook was run using \texttt{run-hook}.

Note that the C functions in the following entries are for handling \textit{Scheme-level} hooks in C. There are also \textit{C-level} hooks which have their own interface (see Section 6.11.6.3 [C Hooks], page 291).

\begin{verbatim}
make-hook [n_args] [Scheme Procedure]
scm_make_hook (n_args) [C Function]
Create a hook for storing procedure of arity \texttt{n_args}. \texttt{n_args} defaults to zero. The returned value is a hook object to be used with the other hook procedures.

hook? x [Scheme Procedure]
scm_hook_p (x) [C Function]
Return \#t if \texttt{x} is a hook, \#f otherwise.

hook-empty? hook [Scheme Procedure]
scm_hook_empty_p (hook) [C Function]
Return \#t if \texttt{hook} is an empty hook, \#f otherwise.

add-hook! hook proc [append_p] [Scheme Procedure]
scm_add_hook_x (hook, proc, append_p) [C Function]
Add the procedure \texttt{proc} to the hook \texttt{hook}. The procedure is added to the end if \texttt{append_p} is true, otherwise it is added to the front. The return value of this procedure is not specified.

remove-hook! hook proc [Scheme Procedure]
scm_remove_hook_x (hook, proc) [C Function]
Remove the procedure \texttt{proc} from the hook \texttt{hook}. The return value of this procedure is not specified.

reset-hook! hook [Scheme Procedure]
scm_reset_hook_x (hook) [C Function]
Remove all procedures from the hook \texttt{hook}. The return value of this procedure is not specified.

hook->list hook [Scheme Procedure]
scm_hook_to_list (hook) [C Function]
Convert the procedure list of \texttt{hook} to a list.
\end{verbatim}
run-hook hook arg ...

scm_run_hook (hook, args)

Apply all procedures from the hook hook to the arguments arg .... The order of the procedure application is first to last. The return value of this procedure is not specified.

If, in C code, you are certain that you have a hook object and well formed argument list for that hook, you can also use scm_c_run_hook, which is identical to scm_run_hook but does no type checking.

void scm_c_run_hook (SCM hook, SCM args)

The same as scm_run_hook but without any type checking to confirm that hook is actually a hook object and that args is a well-formed list matching the arity of the hook.

For C code, SCM_HOOKP is a faster alternative to scm_hook_p:

int SCM_HOOKP (x)

Return 1 if x is a Scheme-level hook, 0 otherwise.

6.11.6.3 Hooks For C Code.

The hooks already described are intended to be populated by Scheme-level procedures. In addition to this, the Guile library provides an independent set of interfaces for the creation and manipulation of hooks that are designed to be populated by functions implemented in C.

The original motivation here was to provide a kind of hook that could safely be invoked at various points during garbage collection. Scheme-level hooks are unsuitable for this purpose as running them could itself require memory allocation, which would then invoke garbage collection recursively .... However, it is also the case that these hooks are easier to work with than the Scheme-level ones if you only want to register C functions with them. So if that is mainly what your code needs to do, you may prefer to use this interface.

To create a C hook, you should allocate storage for a structure of type scm_t_c_hook and then initialize it using scm_c_hook_init.

scm_t_c_hook

Data type for a C hook. The internals of this type should be treated as opaque.

scm_t_c_hook_type

Enumeration of possible hook types, which are:

SCM_C_HOOK_NORMAL
Type of hook for which all the registered functions will always be called.

SCM_C_HOOK_OR
Type of hook for which the sequence of registered functions will be called only until one of them returns C true (a non-NULL pointer).

SCM_C_HOOK_AND
Type of hook for which the sequence of registered functions will be called only until one of them returns C false (a NULL pointer).
void scm_c_hook_init (scm_t_c_hook *hook, void *hook_data, [C Function]
         scm_t_c_hook_type type)
     Initialize the C hook at memory pointed to by hook. type should be one of the values
     of the scm_t_c_hook_type enumeration, and controls how the hook functions will be
     called. hook_data is a closure parameter that will be passed to all registered hook
     functions when they are called.

     To add or remove a C function from a C hook, use scm_c_hook_add or scm_c_hook_
     remove. A hook function must expect three void * parameters which are, respectively:

      hook_data  The hook closure data that was specified at the time the hook was initialized
                by scm_c_hook_init.

      func_data  The function closure data that was specified at the time that that function was
                registered with the hook by scm_c_hook_add.

      data       The call closure data specified by the scm_c_hook_run call that runs the hook.

scm_t_c_hook_function         [C Type]
     Function type for a C hook function: takes three void * parameters and returns a
     void * result.

void scm_c_hook_add (scm_t_c_hook *hook, [C Function]
         scm_t_c_hook_function func, void *func_data, int appendp)
     Add function func, with function closure data func_data, to the C hook hook. The new
     function is appended to the hook’s list of functions if appendp is non-zero, otherwise
     prepended.

void scm_c_hook_remove (scm_t_c_hook *hook, [C Function]
         scm_t_c_hook_function func, void *func_data)
     Remove function func, with function closure data func_data, from the C hook hook.
     scm_c_hook_remove checks both func and func_data so as to allow for the same func
     being registered multiple times with different closure data.

     Finally, to invoke a C hook, call the scm_c_hook_run function specifying the hook and
     the call closure data for this run:

void * scm_c_hook_run (scm_t_c_hook *hook, void *data) [C Function]
     Run the C hook hook will call closure data data. Subject to the variations for hook
     types SCM_C_HOOK_OR and SCM_C_HOOK_AND, scm_c_hook_run calls hook’s registered
     functions in turn, passing them the hook’s closure data, each function’s closure data,
     and the call closure data.
     scm_c_hook_run’s return value is the return value of the last function to be called.

6.11.6.4 Hooks for Garbage Collection
Whensoever Guile performs a garbage collection, it calls the following hooks in the order
shown.

scm_before_gc_c_hook  [C Hook]
     C hook called at the very start of a garbage collection, after setting scm_gc_running_
p to 1, but before entering the GC critical section.
If garbage collection is blocked because `scm_block_gc` is non-zero, GC exits early soon after calling this hook, and no further hooks will be called.

### scm_before_mark_c_hook

*C Hook*

C hook called before beginning the mark phase of garbage collection, after the GC thread has entered a critical section.

### scm_before_sweep_c_hook

*C Hook*

C hook called before beginning the sweep phase of garbage collection. This is the same as at the end of the mark phase, since nothing else happens between marking and sweeping.

### scm_after_sweep_c_hook

*C Hook*

C hook called after the end of the sweep phase of garbage collection, but while the GC thread is still inside its critical section.

### scm_after_gc_c_hook

*C Hook*

C hook called at the very end of a garbage collection, after the GC thread has left its critical section.

### after-gc-hook

*Scheme Hook*

Scheme hook with arity 0. This hook is run asynchronously (see Section 6.22.3 [Asyncs], page 439) soon after the GC has completed and any other events that were deferred during garbage collection have been processed. (Also accessible from C with the name `scm_after_gc_hook`.)

All the C hooks listed here have type `SCM_C_HOOK_NORMAL`, are initialized with hook closure data NULL, are invoked by `scm_c_hook_run` with call closure data NULL.

The Scheme hook `after-gc-hook` is particularly useful in conjunction with guardians (see Section 6.19.4 [Guardians], page 405). Typically, if you are using a guardian, you want to call the guardian after garbage collection to see if any of the objects added to the guardian have been collected. By adding a thunk that performs this call to `after-gc-hook`, you can ensure that your guardian is tested after every garbage collection cycle.

### 6.11.6.5 Hooks into the Guile REPL

### 6.12 Definitions and Variable Bindings

Scheme supports the definition of variables in different contexts. Variables can be defined at the top level, so that they are visible in the entire program, and variables can be defined locally to procedures and expressions. This is important for modularity and data abstraction.

### 6.12.1 Top Level Variable Definitions

At the top level of a program (i.e., not nested within any other expression), a definition of the form

```scheme
(define a value)
```

defines a variable called `a` and sets it to the value `value`.
If the variable already exists in the current module, because it has already been created by a previous define expression with the same name, its value is simply changed to the new value. In this case, then, the above form is completely equivalent to

(set! a value)

This equivalence means that define can be used interchangeably with set! to change the value of variables at the top level of the REPL or a Scheme source file. It is useful during interactive development when reloading a Scheme file that you have modified, because it allows the define expressions in that file to work as expected both the first time that the file is loaded and on subsequent occasions.

Note, though, that define and set! are not always equivalent. For example, a set! is not allowed if the named variable does not already exist, and the two expressions can behave differently in the case where there are imported variables visible from another module.

define name value

Create a top level variable named name with value value. If the named variable already exists, just change its value. The return value of a define expression is unspecified.

The C API equivalents of define are scm_define and scm_c_define, which differ from each other in whether the variable name is specified as a SCM symbol or as a null-terminated C string.

scm_define (sym, value)

scm_c_define (const char *name, value)

C equivalents of define, with variable name specified either by sym, a symbol, or by name, a null-terminated C string. Both variants return the new or preexisting variable object.

define (when it occurs at top level), scm_define and scm_c_define all create or set the value of a variable in the top level environment of the current module. If there was not already a variable with the specified name belonging to the current module, but a similarly named variable from another module was visible through having been imported, the newly created variable in the current module will shadow the imported variable, such that the imported variable is no longer visible.

Attention: Scheme definitions inside local binding constructs (see Section 6.12.2 [Local Bindings], page 295) act differently (see Section 6.12.3 [Internal Definitions], page 296).

Many people end up in a development style of adding and changing definitions at runtime, building out their program without restarting it. (You can do this using reload-module, the reload REPL command, the load procedure, or even just pasting code into a REPL.) If you are one of these people, you will find that sometimes there are some variables that you don't want to redefine all the time. For these, use define-once.

deфине-onсе name value

Create a top level variable named name with value value, but only if name is not already bound in the current module.

Old Lispers probably know define-once under its Lisp name, defvar.
6.12.2 Local Variable Bindings

As opposed to definitions at the top level, which creates bindings that are visible to all code in a module, it is also possible to define variables which are only visible in a well-defined part of the program. Normally, this part of a program will be a procedure or a subexpression of a procedure.

With the constructs for local binding (\texttt{let}, \texttt{let*}, \texttt{letrec}, and \texttt{letrec*}), the Scheme language has a block structure like most other programming languages since the days of ALGOL 60. Readers familiar to languages like C or Java should already be used to this concept, but the family of \texttt{let} expressions has a few properties which are well worth knowing.

The most basic local binding construct is \texttt{let}.

\begin{verbatim}
let bindings body
  bindings has the form
    ((variable1 init1) ...) 
    that is zero or more two-element lists of a variable and an arbitrary expression each. 
    All variable names must be distinct. 
    A \texttt{let} expression is evaluated as follows. 
    \begin{itemize}
      \item All \texttt{init} expressions are evaluated. 
      \item New storage is allocated for the \texttt{variables}. 
      \item The values of the \texttt{init} expressions are stored into the variables. 
      \item The expressions in \texttt{body} are evaluated in order, and the value of the last expression is returned as the value of the \texttt{let} expression. 
    \end{itemize}
    The \texttt{init} expressions are not allowed to refer to any of the \texttt{variables}. 

    The other binding constructs are variations on the same theme: making new values, binding them to variables, and executing a body in that new, extended lexical context.

let* bindings body
  Similar to \texttt{let}, but the variable bindings are performed sequentially, that means that all \texttt{init} expression are allowed to use the variables defined on their left in the binding list. 
  A \texttt{let*} expression can always be expressed with nested \texttt{let} expressions.

  (let* ((a 1) (b a))
    b)
  \equiv

  (let ((a 1))
    (let ((b a))
      b))

letrec bindings body
  Similar to \texttt{let}, but it is possible to refer to the \texttt{variable} from lambda expression created in any of the \texttt{inits}. That is, procedures created in the \texttt{init} expression can recursively refer to the defined variables.

  (letrec ((even? (lambda (n)
    (if (zero? n)
#t
(odd? (- n 1)))

(odd? (lambda (n)
    (if (zero? n)
        #f
        (even? (- n 1)))))

(even? 88))
⇒ #t

Note that while the init expressions may refer to the new variables, they may not access their values. For example, making the even? function above creates a closure (see Section 3.4 [About Closure], page 26) referencing the odd? variable. But odd? can’t be called until after execution has entered the body.

letrec* bindings body

Similar to letrec, except the init expressions are bound to their variables in order. letrec* thus relaxes the letrec restriction, in that later init expressions may refer to the values of previously bound variables.

(letrec* ((a 42)
    (b (+ a 10))) ;; Illegal access
    (* a b))
;; The behavior of the expression above is unspecified

⇒ 2184

There is also an alternative form of the let form, which is used for expressing iteration. Because of the use as a looping construct, this form (the named let) is documented in the section about iteration (see Section 6.13.4 [while do], page 301)

6.12.3 Internal definitions

A define form which appears inside the body of a lambda, let, let*, letrec, letrec* or equivalent expression is called an internal definition. An internal definition differs from a top level definition (see Section 6.12.1 [Top Level], page 293), because the definition is only visible inside the complete body of the enclosing form. Let us examine the following example.

(let ((frumble "froz"))
    (define banana (lambda () (apple 'peach)))
    (define apple (lambda (x) x))
    (banana))
⇒
peach

Here the enclosing form is a let, so the defines in the let-body are internal definitions. Because the scope of the internal definitions is the complete body of the let-expression,
the `lambda`-expression which gets bound to the variable `banana` may refer to the variable `apple`, even though its definition appears lexically *after* the definition of `banana`. This is because a sequence of internal definition acts as if it were a `letrec*` expression.

```
(let ()
  (define a 1)
  (define b 2)
  (+ a b))
```

is equivalent to

```
(let ()
  (letrec* ((a 1) (b 2))
    (+ a b)))
```

Internal definitions are only allowed at the beginning of the body of an enclosing expression. They may not be mixed with other expressions.

Another noteworthy difference to top level definitions is that within one group of internal definitions all variable names must be distinct. That means where on the top level a second define for a given variable acts like a `set!`, an exception is thrown for internal definitions with duplicate bindings.

As a historical note, it used to be that internal bindings were expanded in terms of `letrec`, not `letrec*`. This was the situation for the R5RS report and before. However with the R6RS, it was recognized that sequential definition was a more intuitive expansion, as in the following case:

```
(let ()
  (define a 1)
  (define b (+ a a))
  (+ a b))
```

Guile decided to follow the R6RS in this regard, and now expands internal definitions using `letrec*`.

### 6.12.4 Querying variable bindings

Guile provides a procedure for checking whether a symbol is bound in the top level environment.

**defined?**  *sym*  *[module]*  

[Scheme Procedure]

```
scm_defined_p (sym, module)
```

Return `#t` if `sym` is defined in the module `module` or the current module when `module` is not specified; otherwise return `#f`.

### 6.12.5 Binding multiple return values

**define-values**  *formals*  *expression*

[Syntax]

The *expression* is evaluated, and the *formals* are bound to the return values in the same way that the formals in a `lambda` expression are matched to the arguments in a procedure call.

```
(define-values (q r) (floor/ 10 3))
(list q r) ⇒ (3 1)
```
(define-values (x . y) (values 1 2 3))
x ⇒ 1
y ⇒ (2 3)

(define-values x (values 1 2 3))
x ⇒ (1 2 3)

6.13 Controlling the Flow of Program Execution

See Section 5.4.3 [Control Flow], page 68, for a discussion of how the more general control flow of Scheme affects C code.

6.13.1 Sequencing and Splicing

As an expression, the begin syntax is used to evaluate a sequence of sub-expressions in order. Consider the conditional expression below:

(if (> x 0)
  (begin (display "greater") (newline)))

If the test is true, we want to display “greater” to the current output port, then display a newline. We use begin to form a compound expression out of this sequence of sub-expressions.

begin expr ...  [syntax]
The expression(s) are evaluated in left-to-right order and the value of the last expression is returned as the value of the begin-expression. This expression type is used when the expressions before the last one are evaluated for their side effects.

The begin syntax has another role in definition context (see Section 6.12.3 [Internal Definitions], page 296). A begin form in a definition context splices its subforms into its place. For example, consider the following procedure:

(define (make-seal)
  (define-sealant seal open)
  (values seal open))

Let us assume the existence of a define-sealant macro that expands out to some definitions wrapped in a begin, like so:

(define (make-seal)
  (begin
    (define seal-tag
      (list 'seal))
    (define (seal x)
      (cons seal-tag x))
    (define (sealed? x)
      (and (pair? x) (eq? (car x) seal-tag)))
    (define (open x)
      (if (sealed? x)
        (cdr x)
        (error "Expected a sealed value:" x))))
Here, because the `begin` is in definition context, its subforms are spliced into the place of the `begin`. This allows the definitions created by the macro to be visible to the following expression, the `values` form.

It is a fine point, but splicing and sequencing are different. It can make sense to splice zero forms, because it can make sense to have zero internal definitions before the expressions in a procedure or lexical binding form. However, it does not make sense to have a sequence of zero expressions, because in that case it would not be clear what the value of the sequence would be, because in a sequence of zero expressions, there can be no last value. Sequencing zero expressions is an error.

It would be more elegant in some ways to eliminate splicing from the Scheme language, and without macros (see Section 6.10 [Macros], page 261), that would be a good idea. But it is useful to be able to write macros that expand out to multiple definitions, as in `define-sealant` above, so Scheme abuses the `begin` form for these two tasks.

### 6.13.2 Simple Conditional Evaluation

Guile provides three syntactic constructs for conditional evaluation. `if` is the normal if-then-else expression (with an optional else branch), `cond` is a conditional expression with multiple branches and `case` branches if an expression has one of a set of constant values.

```scheme
if test consequent [alternate]                     [syntax]
```

All arguments may be arbitrary expressions. First, `test` is evaluated. If it returns a true value, the expression `consequent` is evaluated and `alternate` is ignored. If `test` evaluates to `#f`, `alternate` is evaluated instead. The values of the evaluated branch (`consequent` or `alternate`) are returned as the values of the `if` expression.

When `alternate` is omitted and the `test` evaluates to `#f`, the value of the expression is not specified.

When you go to write an `if` without an alternate (a one-armed `if`), part of what you are expressing is that you don’t care about the return value (or values) of the expression. As such, you are more interested in the effect of evaluating the consequent expression. (By convention, we use the word `statement` to refer to an expression that is evaluated for effect, not for value).

In such a case, it is considered more clear to express these intentions with these special forms, `when` and `unless`. As an added bonus, these forms accept multiple statements to evaluate, which are implicitly wrapped in a `begin`.

```scheme
when test statement1 statement2 ...               [Scheme Syntax]
unless test statement1 statement2 ...             [Scheme Syntax]
```

The actual definitions of these forms are in many ways their most clear documentation:

```scheme
(define-syntax-rule (when test stmt stmt* ...))
  (if test (begin stmt stmt* ...)))
```

```scheme
(define-syntax-rule (unless condition stmt stmt* ...))
  (if (not test) (begin stmt stmt* ...)))
```

That is to say, `when` evaluates its consequent statements in order if `test` is true. `unless` is the opposite: it evaluates the statements if `test` is false.
cond clause1 clause2 ...  
Each cond-clause must look like this:

(test expression ...)

where test and expression are arbitrary expressions, or like this

(test => expression)

where expression must evaluate to a procedure.

The tests of the clauses are evaluated in order and as soon as one of them evaluates to a true value, the corresponding expressions are evaluated in order and the last value is returned as the value of the cond-expression. For the => clause type, expression is evaluated and the resulting procedure is applied to the value of test. The result of this procedure application is then the result of the cond-expression.

One additional cond-clause is available as an extension to standard Scheme:

(test guard => expression)

where guard and expression must evaluate to procedures. For this clause type, test may return multiple values, and cond ignores its boolean state; instead, cond evaluates guard and applies the resulting procedure to the value(s) of test, as if guard were the consumer argument of call-with-values. If the result of that procedure call is a true value, it evaluates expression and applies the resulting procedure to the value(s) of test, in the same manner as the guard was called.

The test of the last clause may be the symbol else. Then, if none of the preceding tests is true, the expressions following the else are evaluated to produce the result of the cond-expression.

case key clause1 clause2 ...  
key may be any expression, and the clauses must have the form

((datum1 ...) expr1 expr2 ...)

or

((datum1 ...) => expression)

and the last clause may have the form

(else expr1 expr2 ...)

or

(else => expression)

All datums must be distinct. First, key is evaluated. The result of this evaluation is compared against all datum values using eqv?. When this comparison succeeds, the expression(s) following the datum are evaluated from left to right, returning the value of the last expression as the result of the case expression.

If the key matches no datum and there is an else-clause, the expressions following the else are evaluated. If there is no such clause, the result of the expression is unspecified.

For the => clause types, expression is evaluated and the resulting procedure is applied to the value of key. The result of this procedure application is then the result of the case-expression.
6.13.3 Conditional Evaluation of a Sequence of Expressions

`and` and `or` evaluate all their arguments in order, similar to `begin`, but evaluation stops as soon as one of the expressions evaluates to false or true, respectively.

```scheme
and expr . . . [syntax]
Evaluate the exprs from left to right and stop evaluation as soon as one expression evaluates to `#f`; the remaining expressions are not evaluated. The value of the last evaluated expression is returned. If no expression evaluates to `#f`, the value of the last expression is returned.
If used without expressions, `#t` is returned.
```

```scheme
or expr . . . [syntax]
Evaluate the exprs from left to right and stop evaluation as soon as one expression evaluates to a true value (that is, a value different from `#f`); the remaining expressions are not evaluated. The value of the last evaluated expression is returned. If all expressions evaluate to `#f`, `#f` is returned.
If used without expressions, `#f` is returned.
```

6.13.4 Iteration mechanisms

Scheme has only few iteration mechanisms, mainly because iteration in Scheme programs is normally expressed using recursion. Nevertheless, R5RS defines a construct for programming loops, calling `do`. In addition, Guile has an explicit looping syntax called `while`.

```scheme
do ((variable init [step]) . . .) (test expr . . .) body . . . [syntax]
Bind variables and evaluate body until test is true. The return value is the last expr after test, if given. A simple example will illustrate the basic form,
(do ((i 1 (1+ i)))
  ((> i 4))
  (display i))
⊢ 1234
Or with two variables and a final return value,
(do ((i 1 (1+ i))
   (p 3 (* 3 p)))
  ((> i 4)
   p)
  (format #t "3**~s is ~s\n" i p))
⊢
3**1 is 3
3**2 is 9
3**3 is 27
3**4 is 81
⇒
789
```

The `variable` bindings are established like a `let`, in that the expressions are all evaluated and then all bindings made. When iterating, the optional `step` expressions are evaluated with the previous bindings in scope, then new bindings all made.
The `test` expression is a termination condition. Looping stops when the `test` is true. It’s evaluated before running the `body` each time, so if it’s true the first time then `body` is not run at all.

The optional `expr`s after the `test` are evaluated at the end of looping, with the final `variable` bindings available. The last `expr` gives the return value, or if there are no `expr`s the return value is unspecified.

Each iteration establishes bindings to fresh locations for the `variables`, like a new `let` for each iteration. This is done for `variables` without `step` expressions too. The following illustrates this, showing how a new `i` is captured by the `lambda` in each iteration (see Section 3.4 [The Concept of Closure], page 26).

```scheme
(define lst '())
(do ((i 1 (1+ i)))
   ((> i 4))
  (set! lst (cons (lambda () i) lst)))
(map (lambda (proc) (proc)) lst)
⇒ (4 3 2 1)
```

**while cond body ...**

Run a loop executing the `body` forms while `cond` is true. `cond` is tested at the start of each iteration, so if it’s `#f` the first time then `body` is not executed at all.

Within `while`, two extra bindings are provided, they can be used from both `cond` and `body`.

```scheme
break break-arg ...
```

Break out of the `while` form.

```scheme
continue
```

Abandon the current iteration, go back to the start and test `cond` again, etc.

If the loop terminates normally, by the `cond` evaluating to `#f`, then the `while` expression as a whole evaluates to `#f`. If it terminates by a call to `break` with some number of arguments, those arguments are returned from the `while` expression, as multiple values. Otherwise if it terminates by a call to `break` with no arguments, then return value is `#t`.

```scheme
(while #f (error "not reached")) ⇒ #f
(while #t (break)) ⇒ #t
(while #t (break 1 2 3)) ⇒ 1 2 3
```

Each `while` form gets its own `break` and `continue` procedures, operating on that `while`. This means when loops are nested the outer `break` can be used to escape all the way out. For example,

```scheme
(while (test1)
  (let ((outer-break break))
    (while (test2)
      (if (something)
        (outer-break #f))
      ...)))
```
Note that each **break** and **continue** procedure can only be used within the dynamic extent of its **while**. Outside the **while** their behaviour is unspecified.

Another very common way of expressing iteration in Scheme programs is the use of the so-called **named let**.

Named let is a variant of **let** which creates a procedure and calls it in one step. Because of the newly created procedure, named let is more powerful than **do**—it can be used for iteration, but also for arbitrary recursion.

```
(let variable bindings body)  [syntax]
```

For the definition of **bindings** see the documentation about **let** (see Section 6.12.2 [Local Bindings], page 295).

Named **let** works as follows:

- A new procedure which accepts as many arguments as are in **bindings** is created and bound locally (using **let**) to **variable**. The new procedure’s formal argument names are the name of the **variables**.
- The **body** expressions are inserted into the newly created procedure.
- The procedure is called with the **init** expressions as the formal arguments.

The next example implements a loop which iterates (by recursion) 1000 times.

```
(let lp ((x 1000))
  (if (positive? x)
      (lp (- x 1))
    x))
⇒
0
```

### 6.13.5 Prompts

Prompts are control-flow barriers between different parts of a program. In the same way that a user sees a shell prompt (e.g., the Bash prompt) as a barrier between the operating system and her programs, Scheme prompts allow the Scheme programmer to treat parts of programs as if they were running in different operating systems.

We use this roundabout explanation because, unless you’re a functional programming junkie, you probably haven’t heard the term, “delimited, composable continuation”. That’s OK; it’s a relatively recent topic, but a very useful one to know about.

#### 6.13.5.1 Prompt Primitives

Guile’s primitive delimited control operators are **call-with-prompt** and **abort-to-prompt**.

```
call-with-prompt tag thunk handler  [Scheme Procedure]
```

Set up a prompt, and call **thunk** within that prompt.

During the dynamic extent of the call to **thunk**, a prompt named **tag** will be present in the dynamic context, such that if a user calls **abort-to-prompt** (see below) with that tag, control rewinds back to the prompt, and the **handler** is run.

**handler** must be a procedure. The first argument to **handler** will be the state of the computation begun when **thunk** was called, and ending with the call to **abort-to-prompt**. The remaining arguments to **handler** are those passed to **abort-to-prompt**.
**make-prompt-tag** [stem]  
Make a new prompt tag. A prompt tag is simply a unique object. Currently, a prompt tag is a fresh pair. This may change in some future Guile version.

**default-prompt-tag**  
Return the default prompt tag. Having a distinguished default prompt tag allows some useful prompt and abort idioms, discussed in the next section. Note that default-prompt-tag is actually a parameter, and so may be dynamically rebound using parameterize. See Section 6.13.12 [Parameters], page 324.

**abort-to-prompt** tag val1 val2 ...  
Unwind the dynamic and control context to the nearest prompt named tag, also passing the given values.

C programmers may recognize call-with-prompt and abort-to-prompt as a fancy kind of setjmp and longjmp, respectively. Prompts are indeed quite useful as non-local escape mechanisms. Guile’s catch and throw are implemented in terms of prompts. Prompts are more convenient than longjmp, in that one has the opportunity to pass multiple values to the jump target.

Also unlike longjmp, the prompt handler is given the full state of the process that was aborted, as the first argument to the prompt’s handler. That state is the continuation of the computation wrapped by the prompt. It is a delimited continuation, because it is not the whole continuation of the program; rather, just the computation initiated by the call to call-with-prompt.

The continuation is a procedure, and may be reinstated simply by invoking it, with any number of values. Here’s where things get interesting, and complicated as well. Besides being described as delimited, continuations reified by prompts are also composable, because invoking a prompt-saved continuation composes that continuation with the current one.

Imagine you have saved a continuation via call-with-prompt:

```scheme
(define cont
  (call-with-prompt
    ;; tag
    'foo
    ;; thunk
    (lambda ()
      (+ 34 (abort-to-prompt 'foo)))
    ;; handler
    (lambda (k) k)))
```

The resulting continuation is the addition of 34. It’s as if you had written:

```scheme
(define cont
  (lambda (x)
    (+ 34 x)))
```

So, if we call cont with one numeric value, we get that number, incremented by 34:

```scheme
(cont 8)
⇒ 42
(* 2 (cont 8))
```
The last example illustrates what we mean when we say, "composes with the current continuation". We mean that there is a current continuation – some remaining things to compute, like \((\text{lambda} \ (x) \ (* \ x \ 2))\) – and that calling the saved continuation doesn’t wipe out the current continuation, it composes the saved continuation with the current one.

We’re belaboring the point here because traditional Scheme continuations, as discussed in the next section, aren’t composable, and are actually less expressive than continuations captured by prompts. But there’s a place for them both.

Before moving on, we should mention that if the handler of a prompt is a \texttt{lambda} expression, and the first argument isn’t referenced, an abort to that prompt will not cause a continuation to be reified. This can be an important efficiency consideration to keep in mind.

One example where this optimization matters is escape continuations. Escape continuations are delimited continuations whose only use is to make a non-local exit—i.e., to escape from the current continuation. A common use of escape continuations is when throwing an exception (see Section 6.13.8 [Exceptions], page 311).

The constructs below are syntactic sugar atop prompts to simplify the use of escape continuations.

\begin{verbatim}
call-with-escape-continuation proc
\end{verbatim}

\texttt{[Scheme Procedure]}

\begin{verbatim}
call/ec proc
\end{verbatim}

\texttt{[Scheme Procedure]}

Call \texttt{proc} with an escape continuation.

In the example below, the \texttt{return} continuation is used to escape the continuation of the call to \texttt{fold}.

\begin{verbatim}
(\use-modules (ice-9 control)
 (srfi srfi-1))

(define (prefix x lst)
    ;; Return all the elements before the first occurrence
    ;; of X in LST.
    (call/ec
        (\lambda (return)
            (fold (\lambda (element prefix)
                (if (equal? element x)
                    (\return (reverse prefix)) ; escape ‘fold’
                        (\cons element prefix))
                '()
                    lst))))

    (\prefix 'a '0 1 2 a 3 4 5))
\Rightarrow (0 1 2)
\end{verbatim}

\begin{verbatim}
let-escape-continuation k body ...
\end{verbatim}

\texttt{[Scheme Syntax]}

\begin{verbatim}
let/ec k body ...
\end{verbatim}

\texttt{[Scheme Syntax]}

Bind \texttt{k} within \texttt{body} to an escape continuation.

This is equivalent to \((\text{call/ec (lambda (k) body ...))}\).
Additionally there is another helper primitive exported by \texttt{(ice-9 control)}, so load up that module for \texttt{suspendable-continuation}?:

\begin{quote}
\texttt{(use-modules (ice-9 control))}
\end{quote}

\texttt{suspendable-continuation? tag} \hfill \textbf{[Scheme Procedure]}

Return \#t if a call to \texttt{abort-to-prompt} with the prompt tag \texttt{tag} would produce a delimited continuation that could be resumed later.

Almost all continuations have this property. The exception is where some code between the \texttt{call-with-prompt} and the \texttt{abort-to-prompt} recursed through C for some reason, the \texttt{abort-to-prompt} will succeed but any attempt to resume the continuation (by calling it) would fail. This is because composing a saved continuation with the current continuation involves relocating the stack frames that were saved from the old stack onto a (possibly) new position on the new stack, and Guile can only do this for stack frames that it created for Scheme code, not stack frames created by the C compiler. It’s a bit gnarly but if you stick with Scheme, you won’t have any problem.

If no prompt is found with the given tag, this procedure just returns \#f.

\textbf{6.13.5.2 Shift, Reset, and All That}

There is a whole zoo of delimited control operators, and as it does not seem to be a bounded set, Guile implements support for them in a separate module:

\begin{quote}
\texttt{(use-modules (ice-9 control))}
\end{quote}

Firstly, we have a helpful abbreviation for the \texttt{call-with-prompt} operator.

\begin{quote}
\texttt{\% expr} \hfill \textbf{[Scheme Syntax]}
\texttt{\% expr handler} \hfill \textbf{[Scheme Syntax]}
\texttt{\% tag expr handler} \hfill \textbf{[Scheme Syntax]}
\end{quote}

Evaluate \texttt{expr} in a prompt, optionally specifying a tag and a handler. If no tag is given, the default prompt tag is used.

If no handler is given, a default handler is installed. The default handler accepts a procedure of one argument, which will called on the captured continuation, within a prompt.

Sometimes it’s easier just to show code, as in this case:

\begin{quote}
\texttt{(define (default-prompt-handler k proc)}
\texttt{(\% (default-prompt-tag)}
\texttt{(proc k)}
\texttt{(default-prompt-handler))} \hfill \textbf{[Scheme Syntax]}
\end{quote}

The \% symbol is chosen because it looks like a prompt.

Likewise there is an abbreviation for \texttt{abort-to-prompt}, which assumes the default prompt tag:

\texttt{abort val1 val2 \ldots} \hfill \textbf{[Scheme Procedure]}

Abort to the default prompt tag, passing \texttt{val1 val2 \ldots} to the handler.
As mentioned before, (ice-9 control) also provides other delimited control operators. This section is a bit technical, and first-time users of delimited continuations should probably come back to it after some practice with %.

Still here? So, when one implements a delimited control operator like call-with-prompt, one needs to make two decisions. Firstly, does the handler run within or outside the prompt? Having the handler run within the prompt allows an abort inside the handler to return to the same prompt handler, which is often useful. However it prevents tail calls from the handler, so it is less general.

Similarly, does invoking a captured continuation reinstate a prompt? Again we have the tradeoff of convenience versus proper tail calls.

These decisions are captured in the Felleisen F operator. If neither the continuations nor the handlers implicitly add a prompt, the operator is known as –F–. This is the case for Guile’s call-with-prompt and abort-to-prompt.

If both continuation and handler implicitly add prompts, then the operator is +F+. shift and reset are such operators.

`reset body1 body2 ...`  \[Scheme Syntax\]

Establish a prompt, and evaluate body1 body2 ... within that prompt.

The prompt handler is designed to work with shift, described below.

`shift cont body1 body2 ...`  \[Scheme Syntax\]

Abort to the nearest reset, and evaluate body1 body2 ... in a context in which the captured continuation is bound to cont.

As mentioned above, taken together, the body1 body2 ... expressions and the invocations of cont implicitly establish a prompt.

Interested readers are invited to explore Oleg Kiselyov’s wonderful web site at http://okmij.org/ftp/, for more information on these operators.

### 6.13.6 Continuations

A “continuation” is the code that will execute when a given function or expression returns. For example, consider

```scheme
(define (foo)
  (display "hello\n")
  (display (bar)) (newline)
  (exit))
```

The continuation from the call to bar comprises a display of the value returned, a newline and an exit. This can be expressed as a function of one argument.

```scheme
(lambda (r)
  (display r) (newline)
  (exit))
```

In Scheme, continuations are represented as special procedures just like this. The special property is that when a continuation is called it abandons the current program location and jumps directly to that represented by the continuation.
A continuation is like a dynamic label, capturing at run-time a point in program execution, including all the nested calls that have lead to it (or rather the code that will execute when those calls return).

Continuations are created with the following functions.

```
call-with-current-continuation proc
```

[Scheme Procedure]

Capture the current continuation and call \((\text{proc } \text{cont})\) with it. The return value is the value returned by \(\text{proc}\), or when \((\text{cont } \text{value})\) is later invoked, the return is the \text{value} passed.

Normally \text{cont} should be called with one argument, but when the location resumed is expecting multiple values (see Section 6.13.7 [Multiple Values], page 309) then they should be passed as multiple arguments, for instance \((\text{cont } x \ y \ z)\).

\text{cont} may only be used from the same side of a continuation barrier as it was created (see Section 6.13.14 [Continuation Barriers], page 328), and in a multi-threaded program only from the thread in which it was created.

The call to \text{proc} is not part of the continuation captured, it runs only when the continuation is created. Often a program will want to store \text{cont} somewhere for later use; this can be done in \text{proc}.

The \text{call} in the name \text{call-with-current-continuation} refers to the way a call to \text{proc} gives the newly created continuation. It’s not related to the way a call is used later to invoke that continuation.

\text{call/cc} is an alias for \text{call-with-current-continuation}. This is in common use since the latter is rather long.

Here is a simple example,

```
(define kont #f)
(format #t "the return is ~a\n"
 (call/cc (lambda (k)
          (set! kont k)
          1)))
⇒ the return is 1

(kont 2)
⇒ the return is 2
```

call/cc captures a continuation in which the value returned is going to be displayed by \text{format}. The \text{lambda} stores this in \text{kont} and gives an initial return 1 which is displayed. The later invocation of \text{kont} resumes the captured point, but this time returning 2, which is displayed.

When Guile is run interactively, a call to \text{format} like this has an implicit return back to the read-eval-print loop. call/cc captures that like any other return, which is why interactively \text{kont} will come back to read more input.

C programmers may note that call/cc is like setjmp in the way it records at runtime a point in program execution. A call to a continuation is like a longjmp in that it abandons
the present location and goes to the recorded one. Like longjmp, the value passed to the continuation is the value returned by call/cc on resuming there. However longjmp can only go up the program stack, but the continuation mechanism can go anywhere.

When a continuation is invoked, call/cc and subsequent code effectively “returns” a second time. It can be confusing to imagine a function returning more times than it was called. It may help instead to think of it being stealthily re-entered and then program flow going on as normal.

dynamic-wind (see Section 6.13.10 [Dynamic Wind], page 317) can be used to ensure setup and cleanup code is run when a program locus is resumed or abandoned through the continuation mechanism.

Continuations are a powerful mechanism, and can be used to implement almost any sort of control structure, such as loops, coroutines, or exception handlers.

However the implementation of continuations in Guile is not as efficient as one might hope, because Guile is designed to cooperate with programs written in other languages, such as C, which do not know about continuations. Basically continuations are captured by a block copy of the stack, and resumed by copying back.

For this reason, continuations captured by call/cc should be used only when there is no other simple way to achieve the desired result, or when the elegance of the continuation mechanism outweighs the need for performance.

Escapes upwards from loops or nested functions are generally best handled with prompts (see Section 6.13.5 [Prompts], page 303). Coroutines can be efficiently implemented with cooperating threads (a thread holds a full program stack but doesn’t copy it around the way continuations do).

6.13.7 Returning and Accepting Multiple Values

Scheme allows a procedure to return more than one value to its caller. This is quite different to other languages which only allow single-value returns. Returning multiple values is different from returning a list (or pair or vector) of values to the caller, because conceptually not one compound object is returned, but several distinct values.

The primitive procedures for handling multiple values are values and call-with-values. values is used for returning multiple values from a procedure. This is done by placing a call to values with zero or more arguments in tail position in a procedure body. call-with-values combines a procedure returning multiple values with a procedure which accepts these values as parameters.

values arg ...  
scm_values (args)  

[Scheme Procedure]  
[C Function]  

Delivers all of its arguments to its continuation. Except for continuations created by the call-with-values procedure, all continuations take exactly one value. The effect of passing no value or more than one value to continuations that were not created by call-with-values is unspecified.

For scm_values, args is a list of arguments and the return is a multiple-values object which the caller can return. In the current implementation that object shares structure with args, so args should not be modified subsequently.
SCM scm_c_values (SCM *base, size_t n)  
scm_c_values is an alternative to scm_values. It creates a new values object, and copies into it the n values starting from base.
Currently this creates a list and passes it to scm_values, but we expect that in the future we will be able to use a more efficient representation.

size_t scm_c_nvalues (SCM obj)  
If obj is a multiple-values object, returns the number of values it contains. Otherwise returns 1.

SCM scm_c_value_ref (SCM obj, size_t idx)  
Returns the value at the position specified by idx in obj. Note that obj will ordinarily be a multiple-values object, but it need not be. Any other object represents a single value (itself), and is handled appropriately.

call-with-values producer consumer  
Calls its producer argument with no values and a continuation that, when passed some values, calls the consumer procedure with those values as arguments. The continuation for the call to consumer is the continuation of the call to call-with-values.

```
(called-with-values (lambda () (values 4 5))
  (lambda (a b) b))
⇒ 5
```

```
(called-with-values * -)
⇒ -1
```

In addition to the fundamental procedures described above, Guile has a module which exports a syntax called receive, which is much more convenient. This is in the (ice-9 receive) and is the same as specified by SRFI-8 (see Section 7.5.7 [SRFI-8], page 601).

```
(use-modules (ice-9 receive))
```

receive formals expr body . . .  
Evaluate the expression expr, and bind the result values (zero or more) to the formal arguments in formals. formals is a list of symbols, like the argument list in a lambda (see Section 6.9.1 [Lambda], page 248). After binding the variables, the expressions in body . . . are evaluated in order, the return value is the result from the last expression.

For example getting results from partition in SRFI-1 (see Section 7.5.3 [SRFI-1], page 578),

```
(receive (odds evens)
  (partition odd? '(7 4 2 8 3))
  (display odds)
  (display " and ")
  (display evens))
⇒ (7 3) and (4 2 8)
6.13.8 Exceptions

A common requirement in applications is to want to jump non-locally from the depths of a computation back to, say, the application’s main processing loop. Usually, the place that is the target of the jump is somewhere in the calling stack of procedures that called the procedure that wants to jump back. For example, typical logic for a key press driven application might look something like this:

```scheme
main-loop:
  read the next key press and call dispatch-key

dispatch-key:
  lookup the key in a keymap and call an appropriate procedure, say find-file

find-file:
  interactively read the required file name, then call find-specified-file

find-specified-file:
  check whether file exists; if not, jump back to main-loop
...```

The jump back to `main-loop` could be achieved by returning through the stack one procedure at a time, using the return value of each procedure to indicate the error condition, but Guile (like most modern programming languages) provides an additional mechanism called exception handling that can be used to implement such jumps much more conveniently.

6.13.8.1 Exception Terminology

There are several variations on the terminology for dealing with non-local jumps. It is useful to be aware of them, and to realize that they all refer to the same basic mechanism.

- Actually making a non-local jump may be called raising an exception, raising a signal, throwing an exception or doing a long jump. When the jump indicates an error condition, people may talk about signalling, raising or throwing an error.
- Handling the jump at its target may be referred to as catching or handling the exception, signal or, where an error condition is involved, error.

Where signal and signalling are used, special care is needed to avoid the risk of confusion with POSIX signals.

This manual prefers to speak of throwing and catching exceptions, since this terminology matches the corresponding Guile primitives.

The exception mechanism described in this section has connections with delimited continuations (see Section 6.13.5 [Prompts], page 303). In particular, throwing an exception is akin to invoking an escape continuation (see Section 6.13.5.1 [Prompt Primitives], page 303).

6.13.8.2 Catching Exceptions

catch is used to set up a target for a possible non-local jump. The arguments of a catch expression are a key, which restricts the set of exceptions to which this catch applies, a
thunk that specifies the code to execute and one or two handler procedures that say what to do if an exception is thrown while executing the code. If the execution thunk executes normally, which means without throwing any exceptions, the handler procedures are not called at all.

When an exception is thrown using the throw function, the first argument of the throw is a symbol that indicates the type of the exception. For example, Guile throws an exception using the symbol numerical-overflow to indicate numerical overflow errors such as division by zero:

\[
(/ \ 1 \ 0)
\]

⇒

ABORT: (numerical-overflow)

The key argument in a catch expression corresponds to this symbol. key may be a specific symbol, such as numerical-overflow, in which case the catch applies specifically to exceptions of that type; or it may be #t, which means that the catch applies to all exceptions, irrespective of their type.

The second argument of a catch expression should be a thunk (i.e. a procedure that accepts no arguments) that specifies the normal case code. The catch is active for the execution of this thunk, including any code called directly or indirectly by the thunk’s body. Evaluation of the catch expression activates the catch and then calls this thunk.

The third argument of a catch expression is a handler procedure. If an exception is thrown, this procedure is called with exactly the arguments specified by the throw. Therefore, the handler procedure must be designed to accept a number of arguments that corresponds to the number of arguments in all throw expressions that can be caught by this catch.

The fourth, optional argument of a catch expression is another handler procedure, called the pre-unwind handler. It differs from the third argument in that if an exception is thrown, it is called, before the third argument handler, in exactly the dynamic context of the throw expression that threw the exception. This means that it is useful for capturing or displaying the stack at the point of the throw, or for examining other aspects of the dynamic context, such as fluid values, before the context is unwound back to that of the prevailing catch.

\[
\text{catch key thunk handler [pre-unwind-handler]} \quad \text{[Scheme Procedure]}
\]

\[
\text{scm\_catch\_with\_pre\_unwind\_handler (key, thunk, handler, pre\_unwind\_handler)} \quad \text{[C Function]}
\]

\[
\text{scm\_catch (key, thunk, handler)} \quad \text{[C Function]}
\]

Invoke thunk in the dynamic context of handler for exceptions matching key. If thunk throws to the symbol key, then handler is invoked this way:

\[
\text{(handler key args ...)}
\]

key is a symbol or #t.

thunk takes no arguments. If thunk returns normally, that is the return value of catch.

Handler is invoked outside the scope of its own catch. If handler again throws to the same key, a new handler from further up the call chain is invoked.

If the key is #t, then a throw to any symbol will match this call to catch.
If a *pre-unwind-handler* is given and *thunk* throws an exception that matches *key*, Guile calls the *pre-unwind-handler* before unwinding the dynamic state and invoking the main *handler*. *pre-unwind-handler* should be a procedure with the same signature as *handler*, that is, `(lambda (key . args))`. It is typically used to save the stack at the point where the exception occurred, but can also query other parts of the dynamic state at that point, such as fluid values.

A *pre-unwind-handler* can exit either normally or non-locally. If it exits normally, Guile unwinds the stack and dynamic context and then calls the normal (third argument) handler. If it exits non-locally, that exit determines the continuation.

If a handler procedure needs to match a variety of *throw* expressions with varying numbers of arguments, you should write it like this:

```scheme
(lambdas (key . args)
    ...
)
```

The *key* argument is guaranteed always to be present, because a *throw* without a *key* is not valid. The number and interpretation of the *args* varies from one type of exception to another, but should be specified by the documentation for each exception type.

Note that, once the normal (post-unwind) handler procedure is invoked, the catch that led to the handler procedure being called is no longer active. Therefore, if the handler procedure itself throws an exception, that exception can only be caught by another active catch higher up the call stack, if there is one.

```c
SCM scm_c_catch (SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, void *pre_unwind_handler_data);
SCM scm_internal_catch (SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data);
```

The above `scm_catch_with_pre_unwind_handler` and `scm_catch` take Scheme procedures as body and handler arguments. `scm_c_catch` and `scm_internal_catch` are equivalents taking C functions.

*body* is called as *body* (*body_data*) with a catch on exceptions of the given *tag* type. If an exception is caught, *pre_unwind_handler* and *handler* are called as *handler* (*handler_data, key, args*). *key* and *args* are the SCM key and argument list from the *throw*.

*body* and *handler* should have the following prototypes. `scm_t_catch_body` and `scm_t_catch_handler` are pointer typedefs for these.

```
SCM body (void *data);
SCM handler (void *data, SCM key, SCM args);
```

The *body_data* and *handler_data* parameters are passed to the respective calls so an application can communicate extra information to those functions.

If the data consists of an SCM object, care should be taken that it isn’t garbage collected while still required. If the SCM is a local C variable, one way to protect it is to pass a pointer to that variable as the data parameter, since the C compiler will then know the value must be held on the stack. Another way is to use `scm_remember_up_to_here_1` (see Section 5.5.4 [Foreign Object Memory Management], page 77).
6.13.8.3 Throw Handlers

It’s sometimes useful to be able to intercept an exception that is being thrown before the stack is unwound. This could be to clean up some related state, to print a backtrace, or to pass information about the exception to a debugger, for example. The `with-throw-handler` procedure provides a way to do this.

```
with-throw-handler key thunk handler
```

Add `handler` to the dynamic context as a throw handler for key `key`, then invoke `thunk`.

This behaves exactly like `catch`, except that it does not unwind the stack before invoking `handler`. If the `handler` procedure returns normally, Guile rethrows the same exception again to the next innermost catch or throw handler. `handler` may exit nonlocally, of course, via an explicit throw or via invoking a continuation.

Typically `handler` is used to display a backtrace of the stack at the point where the corresponding `throw` occurred, or to save off this information for possible display later.

Not unwinding the stack means that throwing an exception that is handled via a throw handler is equivalent to calling the throw handler handler inline instead of each `throw`, and then omitting the surrounding `with-throw-handler`. In other words,

```
(with-throw-handler 'key
  (lambda () ... (throw 'key args ...) ...)
handler)
```

is mostly equivalent to

```
((lambda () ... (handler 'key args ...) ...))
```

In particular, the dynamic context when `handler` is invoked is that of the site where `throw` is called. The examples are not quite equivalent, because the body of a `with-throw-handler` is not in tail position with respect to the `with-throw-handler`, and if `handler` exits normally, Guile arranges to rethrow the error, but hopefully the intention is clear. (For an introduction to what is meant by dynamic context, See Section 6.13.10 [Dynamic Wind], page 317.)

```
SCM scm_c_with_throw_handler (SCM tag, scm_t_catch_body
  body, void *body_data, scm_t_catch_handler handler, void
  *handler_data, int lazy_catch_p)
```

The above `scm_with_throw_handler` takes Scheme procedures as body (thunk) and handler arguments. `scm_c_with_throw_handler` is an equivalent taking C functions. See `scm_c_catch` (see Section 6.13.8.2 [Catch], page 311) for a description of the parameters, the behaviour however of course follows `with-throw-handler`.

If `thunk` throws an exception, Guile handles that exception by invoking the innermost `catch` or throw handler whose key matches that of the exception. When the innermost thing is a throw handler, Guile calls the specified handler procedure using `(apply handler key args)`. The handler procedure may either return normally or exit non-locally. If it returns normally, Guile passes the exception on to the next innermost `catch` or throw handler. If it exits non-locally, that exit determines the continuation.
The behaviour of a throw handler is very similar to that of a catch expression’s optional pre-unwind handler. In particular, a throw handler’s handler procedure is invoked in the exact dynamic context of the throw expression, just as a pre-unwind handler is. with-throw-handler may be seen as a half-catch: it does everything that a catch would do until the point where catch would start unwinding the stack and dynamic context, but then it rethrows to the next innermost catch or throw handler instead.

Note also that since the dynamic context is not unwound, if a with-throw-handler handler throws to a key that does not match the with-throw-handler expression’s key, the new throw may be handled by a catch or throw handler that is closer to the throw than the first with-throw-handler.

Here is an example to illustrate this behavior:

```scheme
(catch 'a
  (lambda ()
    (with-throw-handler 'b
      (lambda ()
        (catch 'a
          (lambda ()
            (throw 'b)))
        inner-handler))
      (lambda (key . args)
        (throw 'a)))))
  outer-handler)
```

This code will call inner-handler and then continue with the continuation of the inner catch.

### 6.13.8.4 Throwing Exceptions

The throw primitive is used to throw an exception. One argument, the key, is mandatory, and must be a symbol; it indicates the type of exception that is being thrown. Following the key, throw accepts any number of additional arguments, whose meaning depends on the exception type. The documentation for each possible type of exception should specify the additional arguments that are expected for that kind of exception.

```scheme
throw key arg ... [Scheme Procedure]
scm_throw (key, args) [C Function]
```

Invoke the catch form matching key, passing arg . . . to the handler.
key is a symbol. It will match catches of the same symbol or of #t.

If there is no handler at all, Guile prints an error and then exits.

When an exception is thrown, it will be caught by the innermost catch or throw handler that applies to the type of the thrown exception; in other words, whose key is either #t or the same symbol as that used in the throw expression. Once Guile has identified the appropriate catch or throw handler, it handles the exception by applying the relevant handler procedure(s) to the arguments of the throw.

If there is no appropriate catch or throw handler for a thrown exception, Guile prints an error to the current error port indicating an uncaught exception, and then exits. In practice, it is quite difficult to observe this behaviour, because Guile when used interactively installs
a top level catch handler that will catch all exceptions and print an appropriate error message without exiting. For example, this is what happens if you try to throw an unhandled exception in the standard Guile REPL; note that Guile’s command loop continues after the error message:

```
guile> (throw 'badex)
<unnamed port>:3:1: In procedure gsubr-apply ...
<unnamed port>:3:1: unhandled-exception: badex
ABORT: (misc-error)
guile>
```

The default uncaught exception behaviour can be observed by evaluating a throw expression from the shell command line:

```
$ guile -c "(begin (throw 'badex) (display \"here\\n\")\n)"
guile: uncaught throw to badex: ()
$
```

That Guile exits immediately following the uncaught exception is shown by the absence of any output from the display expression, because Guile never gets to the point of evaluating that expression.

### 6.13.8.5 How Guile Implements Exceptions

It is traditional in Scheme to implement exception systems using call-with-current-continuation. Continuations (see Section 6.13.6 [Continuations], page 307) are such a powerful concept that any other control mechanism — including catch and throw — can be implemented in terms of them.

Guile does not implement catch and throw like this, though. Why not? Because Guile is specifically designed to be easy to integrate with applications written in C. In a mixed Scheme/C environment, the concept of continuation must logically include “what happens next” in the C parts of the application as well as the Scheme parts, and it turns out that the only reasonable way of implementing continuations like this is to save and restore the complete C stack.

So Guile’s implementation of call-with-current-continuation is a stack copying one. This allows it to interact well with ordinary C code, but means that creating and calling a continuation is slowed down by the time that it takes to copy the C stack.

The more targeted mechanism provided by catch and throw does not need to save and restore the C stack because the throw always jumps to a location higher up the stack of the code that executes the throw. Therefore Guile implements the catch and throw primitives independently of call-with-current-continuation, in a way that takes advantage of this upwards only nature of exceptions.

### 6.13.9 Procedures for Signaling Errors

Guile provides a set of convenience procedures for signaling error conditions that are implemented on top of the exception primitives just described.

**error**  
```
(Scheme Procedure)
Raise an error with key misc-error and a message constructed by displaying msg and writing arg . . .
```
scm-error key subr message args data  

scm_error_scm (key, subr, message, args, data)  

Raise an error with key key. subr can be a string naming the procedure associated with the error, or #f. message is the error message string, possibly containing "S and "A escapes. When an error is reported, these are replaced by formatting the corresponding members of args: "A (was %s in older versions of Guile) formats using display and "S (was %S) formats using write. data is a list or #f depending on key: if key is system-error then it should be a list containing the Unix errno value; If key is signal then it should be a list containing the Unix signal number; If key is out-of-range, wrong-type-arg, or keyword-argument-error, it is a list containing the bad value; otherwise it will usually be #f.

strerror err  

scm_strerror (err)  

Return the Unix error message corresponding to err, an integer errno value.

When setlocale has been called (see Section 7.2.13 [Locales], page 542), the message is in the language and charset of LC_MESSAGES. (This is done by the C library.)

false-if-exception expr  

Returns the result of evaluating its argument; however if an exception occurs then #f is returned instead.

6.13.10 Dynamic Wind

For Scheme code, the fundamental procedure to react to non-local entry and exits of dynamic contexts is dynamic-wind. C code could use scm_internal_dynamic_wind, but since C does not allow the convenient construction of anonymous procedures that close over lexical variables, this will be, well, inconvenient.

Therefore, Guile offers the functions scm_dynwind_begin and scm_dynwind_end to delimit a dynamic extent. Within this dynamic extent, which is called a dynwind context, you can perform various dynwind actions that control what happens when the dynwind context is entered or left. For example, you can register a cleanup routine with scm_dynwind_unwind_handler that is executed when the context is left. There are several other more specialized dynwind actions as well, for example to temporarily block the execution of asyncs or to temporarily change the current output port. They are described elsewhere in this manual.

Here is an example that shows how to prevent memory leaks.

    /* Suppose there is a function called FOO in some library that you
       would like to make available to Scheme code (or to C code that
       follows the Scheme conventions).

       FOO takes two C strings and returns a new string. When an error has
       occurred in FOO, it returns NULL.
    */

    char *foo (char *s1, char *s2);
/* SCM_FOO interfaces the C function FOO to the Scheme way of life. It takes care to free up all temporary strings in the case of non-local exits. */

SCM
scm_foo (SCM s1, SCM s2)
{
    char *c_s1, *c_s2, *c_res;

    scm_dynwind_begin (0);
    c_s1 = scm_toLocale_string (s1);
    scm_dynwind_unwind_handler (free, c_s1, SCM_F_WIND_EXPLICITLY);
    c_s2 = scm_toLocale_string (s2);
    scm_dynwind_free (c_s2);
    c_res = foo (c_s1, c_s2);
    if (c_res == NULL)
        scm_memory_error ("foo");
    scm_dynwind_end ();

    return scm_takeLocale_string (res);
}

/* Call 'free (c_s1)' when the dynwind context is left. */
scm_dynwind_unwind_handler (free, c_s1, SCM_F_WIND_EXPLICITLY);

/* Same as above, but more concisely. */
scm_dynwind_free (c_s2);

scm_dynwind_end ();

return scm_takeLocale_string (res);
}

(define x 'normal-binding)
⇒ x
(define a-cont
  (call-with-current-continuation
   (lambda (escape)
(let ((old-x x))
  (dynamic-wind
    ;; in-guard:
    ;;
    (lambda () (set! x 'special-binding))
    ;; thunk
    ;;
    (lambda () (display x) (newline)
      (call-with-current-continuation escape)
      (display x) (newline)
      x)
    ;; out-guard:
    ;;
    (lambda () (set! x old-x))))))

;; Prints:
special-binding
;; Evaluates to:
⇒ a-cont
x
⇒ normal-binding
(a-cont #f)
;; Prints:
special-binding
;; Evaluates to:
⇒ a-cont ;; the value of the (define a-cont...)
x
⇒ normal-binding
a-cont
⇒ special-binding

scm_t_dynwind_flags
   [C Type]
This is an enumeration of several flags that modify the behavior of scm_dynwind_begin. The flags are listed in the following table.

SCM_F_DYNWIND_REWINDABLE
The dynamic context is rewindable. This means that it can be reentered non-locally (via the invocation of a continuation). The default is that a dynwind context can not be reentered non-locally.

void scm_dynwind_begin (scm_t_dynwind_flags flags)
   [C Function]
The function scm_dynwind_begin starts a new dynamic context and makes it the 'current' one.

The flags argument determines the default behavior of the context. Normally, use 0. This will result in a context that can not be reentered with a captured continuation. When you are prepared to handle reentries, include SCM_F_DYNWIND_REWINDABLE in flags.
Being prepared for reentry means that the effects of unwind handlers can be undone on reentry. In the example above, we want to prevent a memory leak on non-local exit and thus register an unwind handler that frees the memory. But once the memory is freed, we can not get it back on reentry. Thus reentry can not be allowed.

The consequence is that continuations become less useful when non-reentrant contexts are captured, but you don’t need to worry about that too much.

The context is ended either implicitly when a non-local exit happens, or explicitly with `scm_dynwind_end`. You must make sure that a dynwind context is indeed ended properly. If you fail to call `scm_dynwind_end` for each `scm_dynwind_begin`, the behavior is undefined.

```c
void scm_dynwind_end ()
End the current dynamic context explicitly and make the previous one current.
```

```c
scm_t_wind_flags
This is an enumeration of several flags that modify the behavior of `scm_dynwind_unwind_handler` and `scm_dynwind_rewind_handler`. The flags are listed in the following table.

<table>
<thead>
<tr>
<th>SCM_F_WIND_EXPLICITLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>The registered action is also carried out when the dynwind context is entered or left locally.</td>
</tr>
</tbody>
</table>
```

```c
void scm_dynwind_unwind_handler (void (*func)(void *), void *data, scm_t_wind_flags flags)
void scm_dynwind_unwind_handler_with_scm (void (*func)(SCM), SCM data, scm_t_wind_flags flags)
Arranges for func to be called with data as its arguments when the current context ends implicitly. If flags contains SCM_F_WIND_EXPLICITLY, func is also called when the context ends explicitly with `scm_dynwind_end`.

The function `scm_dynwind_unwind_handler_with_scm` takes care that `data` is protected from garbage collection.
```

```c
void scm_dynwind_rewind_handler (void (*func)(void *), void *data, scm_t_wind_flags flags)
void scm_dynwind_rewind_handler_with_scm (void (*func)(SCM), SCM data, scm_t_wind_flags flags)
Arrange for func to be called with data as its argument when the current context is restarted by rewinding the stack. When flags contains SCM_F_WIND_EXPLICITLY, func is called immediately as well.

The function `scm_dynwind_rewind_handler_with_scm` takes care that `data` is protected from garbage collection.
```

```c
void scm_dynwind_free (void *mem)
Arrange for mem to be freed automatically whenever the current context is exited, whether normally or non-locally. `scm_dynwind_free (mem)` is an equivalent shorthand for `scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`. 
```
6.13.11 Fluids and Dynamic States

A fluid is a variable whose value is associated with the dynamic extent of a function call. In the same way that an operating system runs a process with a given set of current input and output ports (or file descriptors), in Guile you can arrange to call a function while binding a fluid to a particular value. That association between fluid and value will exist during the dynamic extent of the function call.

Fluids are therefore a building block for implementing dynamically scoped variables. Dynamically scoped variables are useful when you want to set a variable to a value during some dynamic extent in the execution of your program and have them revert to their original value when the control flow is outside of this dynamic extent. See the description of with-fluids below for details. This association between fluids, values, and dynamic extents is robust to multiple entries (as when a captured continuation is invoked more than once) and early exits (for example, when throwing exceptions).

Guile uses fluids to implement parameters (see Section 6.13.12 [Parameters], page 324). Usually you just want to use parameters directly. However it can be useful to know what a fluid is and how it works, so that’s what this section is about.

The current set of fluid-value associations can be captured in a dynamic state object. A dynamic extent is simply that: a snapshot of the current fluid-value associations. Guile users can capture the current dynamic state with current-dynamic-state and restore it later via with-dynamic-state or similar procedures. This facility is especially useful when implementing lightweight thread-like abstractions.

New fluids are created with make-fluid and fluid? is used for testing whether an object is actually a fluid. The values stored in a fluid can be accessed with fluid-ref and fluid-set!.

See Section 6.22.2 [Thread Local Variables], page 438, for further notes on fluids, threads, parameters, and dynamic states.

**make-fluid [dflt]**

[Scheme Procedure]

scm_make_fluid ()

[C Function]

scm_make_fluid_with_default (dflt)

[C Function]

Return a newly created fluid, whose initial value is dflt, or #f if dflt is not given. Fluids are objects that can hold one value per dynamic state. That is, modifications to this value are only visible to code that executes with the same dynamic state as the modifying code. When a new dynamic state is constructed, it inherits the values from its parent. Because each thread normally executes with its own dynamic state, you can use fluids for thread local storage.

**make-unbound-fluid**

[Scheme Procedure]

scm_make_unbound_fluid ()

[C Function]

Return a new fluid that is initially unbound (instead of being implicitly bound to some definite value).

**fluid? obj**

[Scheme Procedure]

scm_fluid_p (obj)

[C Function]

Return #t if obj is a fluid; otherwise, return #f.
fluid-ref fluid

scm_fluid_ref (fluid)

[Scheme Procedure]
Return the value associated with fluid in the current dynamic root. If fluid has not been set, then return its default value. Calling fluid-ref on an unbound fluid produces a runtime error.

fluid-set! fluid value

scm_fluid_set_x (fluid, value)

[Scheme Procedure]
[Function]
Set the value associated with fluid in the current dynamic root.

fluid-ref* fluid depth

scm_fluid_ref_star (fluid, depth)

[Scheme Procedure]
[Function]
Return the depthth oldest value associated with fluid in the current thread. If depth equals or exceeds the number of values that have been assigned to fluid, return the default value of the fluid. (fluid-ref* f 0) is equivalent to (fluid-ref f).

fluid-ref* is useful when you want to maintain a stack-like structure in a fluid, such as the stack of current exception handlers. Using fluid-ref* instead of an explicit stack allows any partial continuation captured by call-with-prompt to only capture the bindings made within the limits of the prompt instead of the entire continuation. See Section 6.13.5 [Prompts], page 303, for more on delimited continuations.

fluid-unset! fluid

scm_fluid_unset_x (fluid)

[Scheme Procedure]
[Function]
Disassociate the given fluid from any value, making it unbound.

fluid-bound? fluid

scm_fluid_bound_p (fluid)

[Scheme Procedure]
[Function]
Returns #t if the given fluid is bound to a value, otherwise #f.

with-fluids* temporarily changes the values of one or more fluids, so that the given procedure and each procedure called by it access the given values. After the procedure returns, the old values are restored.

with-fluid* fluid value thunk

scm_with_fluid (fluid, value, thunk)

[Scheme Procedure]
[Function]
Set fluid to value temporarily, and call thunk. thunk must be a procedure with no argument.

with-fluids* fluids values thunk

scm_with_fluids (fluids, values, thunk)

[Scheme Procedure]
[Function]
Set fluids to values temporarily, and call thunk. fluids must be a list of fluids and values must be the same number of their values to be applied. Each substitution is done in the order given. thunk must be a procedure with no argument. It is called inside a dynamic-wind and the fluids are set/restored when control enter or leaves the established dynamic extent.

with-fluids ((fluid value) ...) body1 body2 ...

[Scheme Macro]
Execute body body1 body2 ... while each fluid is set to the corresponding value. Both fluid and value are evaluated and fluid must yield a fluid. The body is executed inside a dynamic-wind and the fluids are set/restored when control enter or leaves the established dynamic extent.
```c
SCM scm_c_with_fluids (SCM fluids, SCM vals, SCM (*cproc)(void *, void *data))

SCM scm_c_with_fluid (SCM fluid, SCM val, SCM (*cproc)(void *), void *data)

The function scm_c_with_fluids is like scm_with_fluids except that it takes a C
function to call instead of a Scheme thunk.

The function scm_c_with_fluid is similar but only allows one fluid to be set instead
of a list.
```

```c
void scm_dynwind_fluid (SCM fluid, SCM val)

This function must be used inside a pair of calls to scm_dynwind_begin and scm_-
dynwind_end (see Section 6.13.10 [Dynamic Wind], page 317). During the dynwind
context, the fluid fluid is set to val.

More precisely, the value of the fluid is swapped with a ‘backup’ value whenever
the dynwind context is entered or left. The backup value is initialized with the val
argument.
```

```c
dynamic-state? obj

scm_dynamic_state_p (obj)

Return #t if obj is a dynamic state object; return #f otherwise.
```

```c
int scm_is_dynamic_state (SCM obj)

Return non-zero if obj is a dynamic state object; return zero otherwise.
```

```c
current-dynamic-state

scm_current_dynamic_state ()

Return a snapshot of the current fluid-value associations as a fresh dynamic state
object.
```

```c
set-current-dynamic-state state

scm_set_current_dynamic_state (state)

Restore the saved fluid-value associations from state, replacing the current fluid-value
associations. Return the current fluid-value associatoins as a dynamic state object,
as in current-dynamic-state.
```

```c
with-dynamic-state state proc

scm_with_dynamic_state (state, proc)

Call proc while the fluid bindings from state have been made current, saving the
current fluid bindings. When control leaves the invocation of proc, restore the saved
bindings, saving instead the fluid bindings from inside the call. If control later re-
enters proc, restore those saved bindings, saving the current bindings, and so on.
```

```c
void scm_dynwind_current_dynamic_state (SCM state)

Set the current dynamic state to state for the current dynwind context. Like
with-dynamic-state, but in terms of Guile’s “dynwind” C API.
```

```c
void * scm_c_with_dynamic_state (SCM state, void *((*func)(void *), void *data))

Like scm_with_dynamic_state, but call func with data.
```
6.13.12 Parameters

Parameters are Guile’s facility for dynamically bound variables.

On the most basic level, a parameter object is a procedure. Calling it with no arguments returns its value. Calling it with one argument sets the value.

```scheme
(define my-param (make-parameter 123))
(my-param) ⇒ 123
(my-param 456)
(my-param) ⇒ 456
```

The `parameterize` special form establishes new locations for parameters, those new locations having effect within the dynamic extent of the `parameterize` body. Leaving restores the previous locations. Re-entering (through a saved continuation) will again use the new locations.

```scheme
(parameterize ((my-param 789))
  (my-param)) ⇒ 789
(my-param) ⇒ 456
```

Parameters are like dynamically bound variables in other Lisp dialects. They allow an application to establish parameter settings (as the name suggests) just for the execution of a particular bit of code, restoring when done. Examples of such parameters might be case-sensitivity for a search, or a prompt for user input.

Global variables are not as good as parameter objects for this sort of thing. Changes to them are visible to all threads, but in Guile parameter object locations are per-thread, thereby truly limiting the effect of `parameterize` to just its dynamic execution.

Passing arguments to functions is thread-safe, but that soon becomes tedious when there’s more than a few or when they need to pass down through several layers of calls before reaching the point they should affect. Introducing a new setting to existing code is often easier with a parameter object than adding arguments.

`make-parameter init [converter]`  
[Scheme Procedure]

Return a new parameter object, with initial value `init`.

If a `converter` is given, then a call `(converter val)` is made for each value set, its return is the value stored. Such a call is made for the `init` initial value too.

A `converter` allows values to be validated, or put into a canonical form. For example,

```scheme
(define my-param (make-parameter 123
  (lambda (val)
    (if (not (number? val))
      (error "must be a number")
      (inexact->exact val))))

(my-param 0.75)
(my-param) ⇒ 3/4
```

`parameterize ((param value) ...) body1 body2 ...`  
[library syntax]

Establish a new dynamic scope with the given `params` bound to new locations and set to the given `values`. `body1 body2 ...` is evaluated in that environment. The value returned is that of last body form.
Each *param* is an expression which is evaluated to get the parameter object. Often this will just be the name of a variable holding the object, but it can be anything that evaluates to a parameter.

The *param* expressions and *value* expressions are all evaluated before establishing the new dynamic bindings, and they’re evaluated in an unspecified order.

For example,

```
(define prompt (make-parameter "Type something: "))
(define (get-input)
  (display (prompt))
...

(parameterize ((prompt "Type a number: "))
  (get-input)
  ...)
```

Parameter objects are implemented using fluids (see Section 6.13.11 [Fluids and Dynamic States], page 321), so each dynamic state has its own parameter locations. That includes the separate locations when outside any *parameterize* form. When a parameter is created it gets a separate initial location in each dynamic state, all initialized to the given *init* value.

New code should probably just use parameters instead of fluids, because the interface is better. But for migrating old code or otherwise providing interoperability, Guile provides the `fluid->parameter` procedure:

```
fluid->parameter fluid [conv]  [Scheme Procedure]
Make a parameter that wraps a fluid.

The value of the parameter will be the same as the value of the fluid. If the parameter is rebound in some dynamic extent, perhaps via *parameterize*, the new value will be run through the optional *conv* procedure, as with any parameter. Note that unlike *make-parameter*, *conv* is not applied to the initial value.

As alluded to above, because each thread usually has a separate dynamic state, each thread has its own locations behind parameter objects, and changes in one thread are not visible to any other. When a new dynamic state or thread is created, the values of parameters in the originating context are copied, into new locations.

Guile’s parameters conform to SRFI-39 (see Section 7.5.27 [SRFI-39], page 624).

### 6.13.13 How to Handle Errors

Error handling is based on *catch* and *throw*. Errors are always thrown with a *key* and four arguments:

- *key*: a symbol which indicates the type of error. The symbols used by libguile are listed below.
- *subr*: the name of the procedure from which the error is thrown, or #f.
- *message*: a string (possibly language and system dependent) describing the error. The tokens ~A and ~S can be embedded within the message: they will be replaced with members of the *args* list when the message is printed. ~A indicates an argument printed using *display*, while ~S indicates an argument printed using *write*. *message* can also
be #f, to allow it to be derived from the key by the error handler (may be useful if the key is to be thrown from both C and Scheme).

- **args**: a list of arguments to be used to expand ~A and ~S tokens in message. Can also be #f if no arguments are required.
- **rest**: a list of any additional objects required. e.g., when the key is ’system-error, this contains the C errno value. Can also be #f if no additional objects are required.

In addition to catch and throw, the following Scheme facilities are available:

```scheme
(display-error frame port subr message args rest)  ; Scheme Procedure
(scm_display_error frame, port, subr, message, args, rest)  ; C Function
```

Display an error message to the output port port. frame is the frame in which the error occurred, subr is the name of the procedure in which the error occurred and message is the actual error message, which may contain formatting instructions. These will format the arguments in the list args accordingly. rest is currently ignored.

The following are the error keys defined by libguile and the situations in which they are used:

- **error-signal**: thrown after receiving an unhandled fatal signal such as SIGSEGV, SIGBUS, SIGFPE etc. The rest argument in the throw contains the coded signal number (at present this is not the same as the usual Unix signal number).
- **system-error**: thrown after the operating system indicates an error condition. The rest argument in the throw contains the errno value.
- **numerical-overflow**: numerical overflow.
- **out-of-range**: the arguments to a procedure do not fall within the accepted domain.
- **wrong-type-arg**: an argument to a procedure has the wrong type.
- **wrong-number-of-args**: a procedure was called with the wrong number of arguments.
- **memory-allocation-error**: memory allocation error.
- **stack-overflow**: stack overflow error.
- **regular-expression-syntax**: errors generated by the regular expression library.
- **misc-error**: other errors.

### 6.13.13.1 C Support

In the following C functions, SUBR and MESSAGE parameters can be NULL to give the effect of #f described above.

```c
SCM scm_error (SCM key, char *subr, char *message, SCM args, SCM rest)  ; C Function
    Throw an error, as per scm-error (see Section 6.13.9 [Error Reporting], page 316).

void scm_syserror (char *subr)  ; C Function
void scm_syserror_msg (char *subr, char *message, SCM args)  ; C Function
    Throw an error with key system-error and supply errno in the rest argument. For scm_syserror the message is generated using strerror.
```

Care should be taken that any code in between the failing operation and the call to these routines doesn’t change errno.
void scm_num_overflow (char *subr)
void scm_out_of_range (char *subr, SCM bad_value)
void scm_wrong_num_args (SCM proc)
void scm_wrong_type_arg (char *subr, int argnum, SCM bad_value)
void scm_wrong_type_arg_msg (char *subr, int argnum, SCM bad_value, const char *expected)
void scm_memory_error (char *subr)
void scm_misc_error (const char *subr, const char *message, SCM args)

Throw an error with the various keys described above.

In scm_wrong_num_args, proc should be a Scheme symbol which is the name of the procedure incorrectly invoked. The other routines take the name of the invoked procedure as a C string.

In scm_wrong_type_arg_msg, expected is a C string describing the type of argument that was expected.

In scm_misc_error, message is the error message string, possibly containing simple-format escapes (see Section 6.14.5 [Simple Output], page 335), and the corresponding arguments in the args list.

6.13.13.2 Signalling Type Errors

Every function visible at the Scheme level should aggressively check the types of its arguments, to avoid misinterpreting a value, and perhaps causing a segmentation fault. Guile provides some macros to make this easier.

void SCM_ASSERT (int test, SCM obj, unsigned int position, const char *subr)
void SCM_ASSERT_TYPE (int test, SCM obj, unsigned int position, const char *subr, const char *expected)

If test is zero, signal a “wrong type argument” error, attributed to the subroutine named subr, operating on the value obj, which is the position’th argument of subr.

In SCM_ASSERT_TYPE, expected is a C string describing the type of argument that was expected.

int SCM_ARG1
int SCM_ARG2
int SCM_ARG3
int SCM_ARG4
int SCM_ARG5
int SCM_ARG6
int SCM_ARG7

One of the above values can be used for position to indicate the number of the argument of subr which is being checked. Alternatively, a positive integer number can be used, which allows to check arguments after the seventh. However, for parameter numbers up to seven it is preferable to use SCM_ARGN instead of the corresponding raw number, since it will make the code easier to understand.
6.13.14 Continuation Barriers

The non-local flow of control caused by continuations might sometimes not be wanted. You can use with-continuation-barrier to erect fences that continuations can not pass.

```scheme
(scm_with_continuation_barrier proc)
```

Call proc and return its result. Do not allow the invocation of continuations that would leave or enter the dynamic extent of the call to with-continuation-barrier. Such an attempt causes an error to be signaled.

Throws (such as errors) that are not caught from within proc are caught by with-continuation-barrier. In that case, a short message is printed to the current error port and #f is returned.

Thus, with-continuation-barrier returns exactly once.

```c
(void * scm_c_with_continuation_barrier (void * (*func) (void *), void *data))
```

Like scm_with_continuation_barrier but call func on data. When an error is caught, NULL is returned.

6.14 Input and Output

6.14.1 Ports

Ports are the way that Guile performs input and output. Guile can read in characters or bytes from an input port, or write them out to an output port. Some ports support both interfaces.

There are a number of different port types implemented in Guile. File ports provide input and output over files, as you might imagine. For example, we might display a string to a file like this:

```scheme
(let ((port (open-output-file "foo.txt")))
  (display "Hello, world!\n" port)
  (close-port port))
```

There are also string ports, for taking input from a string, or collecting output to a string; bytevector ports, for doing the same but using a bytevector as a source or sink of data; and soft ports, for arranging to call Scheme functions to provide input or handle output. See Section 6.14.10 [Port Types], page 341.

Ports should be closed when they are not needed by calling close-port on them, as in the example above. This will make sure that any pending output is successfully written out to disk, in the case of a file port, or otherwise to whatever mutable store is backed by the port. Any error that occurs while writing out that buffered data would also be raised promptly at the close-port, and not later when the port is closed by the garbage collector. See Section 6.14.6 [Buffering], page 336, for more on buffered output.
Closing a port also releases any precious resource the file might have. Usually in Scheme a programmer doesn’t have to clean up after their data structures (see Section 6.19 [Memory Management], page 400), but most systems have strict limits on how many files can be open, both on a per-process and a system-wide basis. A program that uses many files should take care not to hit those limits. The same applies to similar system resources such as pipes and sockets.

Indeed for these reasons the above example is not the most idiomatic way to use ports. It is more common to acquire ports via procedures like \texttt{call-with-output-file}, which handle the \texttt{close-port} automatically:

\begin{verbatim}
(call-with-output-file "foo.txt"
  (lambda (port)
    (display "Hello, world!\n" port)))
\end{verbatim}

Finally, all ports have associated input and output buffers, as appropriate. Buffering is a common strategy to limit the overhead of small reads and writes: without buffering, each character fetched from a file would involve at least one call into the kernel, and maybe more depending on the character and the encoding. Instead, Guile will batch reads and writes into internal buffers. However, sometimes you want to make output on a port show up immediately. See Section 6.14.6 [Buffering], page 336, for more on interfaces to control port buffering.

\begin{verbatim}
port? x
scm_port_p (x)  \[Scheme Procedure\] \[C Function\]
  Return a boolean indicating whether \texttt{x} is a port.

input-port? x
scm_input_port_p (x)  \[Scheme Procedure\] \[C Function\]
  Return \texttt{#t} if \texttt{x} is an input port, otherwise return \texttt{#f}. Any object satisfying this predicate also satisfies \texttt{port?}.

output-port? x
scm_output_port_p (x)  \[Scheme Procedure\] \[C Function\]
  Return \texttt{#t} if \texttt{x} is an output port, otherwise return \texttt{#f}. Any object satisfying this predicate also satisfies \texttt{port?}.

close-port port
scm_close_port (port)  \[Scheme Procedure\] \[C Function\]
  Close the specified port object. Return \texttt{#t} if it successfully closes a port or \texttt{#f} if it was already closed. An exception may be raised if an error occurs, for example when flushing buffered output. See Section 6.14.6 [Buffering], page 336, for more on buffered output. See Section 7.2.2 [Ports and File Descriptors], page 493, for a procedure which can close file descriptors.

port-closed? port
scm_port_closed_p (port)  \[Scheme Procedure\] \[C Function\]
  Return \texttt{#t} if \texttt{port} is closed or \texttt{#f} if it is open.
6.14.2 Binary I/O

Guile’s ports are fundamentally binary in nature: at the lowest level, they work on bytes. This section describes Guile’s core binary I/O operations. See Section 6.14.4 [Textual I/O], page 333, for input and output of strings and characters.

To use these routines, first include the binary I/O module:

```
(use-modules (ice-9 binary-ports))
```

Note that although this module’s name suggests that binary ports are some different kind of port, that’s not the case: all ports in Guile are both binary and textual ports.

### get-u8 port

[Scheme Procedure]

Return an octet read from `port`, an input port, blocking as necessary, or the end-of-file object.

### lookahead-u8 port

[Scheme Procedure]

Like `get-u8` but does not update `port`’s position to point past the octet.

The end-of-file object is unlike any other kind of object: it’s not a pair, a symbol, or anything else. To check if a value is the end-of-file object, use the `eof-object?` predicate.

### eof-object? x

[Scheme Procedure]

Return `#t` if `x` is an end-of-file object, or `#f` otherwise.

Note that unlike other procedures in this module, `eof-object?` is defined in the default environment.

### get-bytevector-n port count

[Scheme Procedure]

Read `count` octets from `port`, blocking as necessary and return a bytevector containing the octets read. If fewer bytes are available, a bytevector smaller than `count` is returned.

### get-bytevector-n! port bv start count

[Scheme Procedure]

Read up to `count` bytes from `port`, blocking as necessary until at least one byte is available or an end-of-file is reached. Store them in `bv` starting at index `start`. Return the number of bytes actually read, or an end-of-file object.

### get-bytevector-some port

[Scheme Procedure]

Read from `port`, blocking as necessary, until bytes are available or an end-of-file is reached. Return either the number of bytes actually read or the end-of-file object.

### get-bytevector-some! port bv start count

[Scheme Procedure]

Read up to `count` bytes from `port`, blocking as necessary until at least one byte is available or an end-of-file is reached. Store them in `bv` starting at index `start`. Return the number of bytes actually read, or an end-of-file object.
get-bytevector-all port

scm_get_bytevector_all (port)

Read from port, blocking as necessary, until the end-of-file is reached. Return either a new bytevector containing the data read or the end-of-file object (if no data were available).

unget-bytevector port bv [start [count]]

scm_unget_bytevector (port, bv, start, count)

Place the contents of bv in port, optionally starting at index start and limiting to count octets, so that its bytes will be read from left-to-right as the next bytes from port during subsequent read operations. If called multiple times, the unread bytes will be read again in last-in first-out order.

To perform binary output on a port, use put-u8 or put-bytevector.

put-u8 port octet

scm_put_u8 (port, octet)

Write octet, an integer in the 0–255 range, to port, a binary output port.

put-bytevector port bv [start [count]]

scm_put_bytevector (port, bv, start, count)

Write the contents of bv to port, optionally starting at index start and limiting to count octets.

6.14.3 Encoding

Textual input and output on Guile ports is layered on top of binary operations. To this end, each port has an associated character encoding that controls how bytes read from the port are converted to characters, and how characters written to the port are converted to bytes.

port-encoding port

scm_port_encoding (port)

Returns, as a string, the character encoding that port uses to interpret its input and output.

set-port-encoding! port enc

scm_set_port_encoding_x (port, enc)

Sets the character encoding that will be used to interpret I/O to port. enc is a string containing the name of an encoding. Valid encoding names are those defined by IANA (http://www.iana.org/assignments/character-sets), for example "UTF-8" or "ISO-8859-1".

When ports are created, they are assigned an encoding. The usual process to determine the initial encoding for a port is to take the value of the %default-port-encoding fluid.

%default-port-encoding

A fluid containing name of the encoding to be used by default for newly created ports (see Section 6.13.11 [Fluids and Dynamic States], page 321). As a special case, the value #f is equivalent to "ISO-8859-1".
The \texttt{%default-port-encoding} itself defaults to the encoding appropriate for the current locale, if \texttt{setlocale} has been called. See Section \ref{Locales} [Locales], page 542, for more on locales and when you might need to call \texttt{setlocale} explicitly.

Some port types have other ways of determining their initial locales. String ports, for example, default to the UTF-8 encoding, in order to be able to represent all characters regardless of the current locale. File ports can optionally sniff their file for a coding declaration; See Section \ref{File Ports} [File Ports], page 341. Binary ports might be initialized to the ISO-8859-1 encoding in which each codepoint between 0 and 255 corresponds to a byte with that value.

Currently, the ports only work with \textit{non-modal} encodings. Most encodings are non-modal, meaning that the conversion of bytes to a string doesn’t depend on its context: the same byte sequence will always return the same string. A couple of modal encodings are in common use, like ISO-2022-JP and ISO-2022-KR, and they are not yet supported.

Each port also has an associated conversion strategy, which determines what to do when a Guile character can’t be converted to the port’s encoded character representation for output. There are three possible strategies: to raise an error, to replace the character with a hex escape, or to replace the character with a substitute character. Port conversion strategies are also used when decoding characters from an input port.

\begin{verbatim}
port-conversion-strategy port

scm_port_conversion_strategy (port)

set-port-conversion-strategy! port sym

scm_set_port_conversion_strategy_x (port, sym)

%default-port-conversion-strategy

\end{verbatim}
As mentioned above, for an output port there are three possible port conversion strategies. The **error** strategy will throw an error when a nonconvertible character is encountered. The **substitute** strategy will replace nonconvertible characters with a question mark (‘?’). Finally the **escape** strategy will print nonconvertible characters as a hex escape, using the escaping that is recognized by Guile’s string syntax. Note that if the port’s encoding is a Unicode encoding, like **UTF-8**, then encoding errors are impossible.

For an input port, the **error** strategy will cause Guile to throw an error if it encounters an invalid encoding, such as might happen if you tried to read **ISO-8859-1** as **UTF-8**. The error is thrown before advancing the read position. The **substitute** strategy will replace the bad bytes with a U+FFFD replacement character, in accordance with Unicode recommendations. When reading from an input port, the **escape** strategy is treated as if it were **error**.

### 6.14.4 Textual I/O

This section describes Guile’s core textual I/O operations on characters and strings. See Section 6.14.2 [Binary I/O], page 330, for input and output of bytes and bytevectors. See Section 6.14.3 [Encoding], page 331, for more on how characters relate to bytes. To read general S-expressions from ports, See Section 6.18.2 [Scheme Read], page 382. See Section 6.18.3 [Scheme Write], page 383, for interfaces that write generic Scheme datums.

To use these routines, first include the textual I/O module:

```scheme
(use-modules (ice-9 textual-ports))
```

Note that although this module’s name suggests that textual ports are some different kind of port, that’s not the case: all ports in Guile are both binary and textual ports.

#### Scheme Procedure: `get-char input-port`

Reads from `input-port`, blocking as necessary, until a complete character is available from `input-port`, or until an end of file is reached.

If a complete character is available before the next end of file, `get-char` returns that character and updates the input port to point past the character. If an end of file is reached before any character is read, `get-char` returns the end-of-file object.

#### Scheme Procedure: `lookahead-char input-port`

The `lookahead-char` procedure is like `get-char`, but it does not update `input-port` to point past the character.

In the same way that it’s possible to "unget" a byte or bytes, it’s possible to "unget" the bytes corresponding to an encoded character.

#### Scheme Procedure: `unget-char port char`

Place character `char` in `port` so that it will be read by the next read operation. If called multiple times, the unread characters will be read again in last-in first-out order.

#### Scheme Procedure: `unget-string port str`

Place the string `str` in `port` so that its characters will be read from left-to-right as the next characters from `port` during subsequent read operations. If called multiple times, the unread characters will be read again in last-in first-out order.
Reading in a character at a time can be inefficient. If it’s possible to perform I/O over multiple characters at a time, via strings, that might be faster.

**get-string-n** input-port count  
[Scheme Procedure]  
The get-string-n procedure reads from input-port, blocking as necessary, until count characters are available, or until an end of file is reached. count must be an exact, non-negative integer, representing the number of characters to be read.

If count characters are available before end of file, get-string-n returns a string consisting of those count characters. If fewer characters are available before an end of file, but one or more characters can be read, get-string-n returns a string containing those characters. In either case, the input port is updated to point just past the characters read. If no characters can be read before an end of file, the end-of-file object is returned.

**get-string-n!** input-port string start count  
[Scheme Procedure]  
The get-string-n! procedure reads from input-port in the same manner as get-string-n. start and count must be exact, non-negative integer objects, with count representing the number of characters to be read. string must be a string with at least $start + count$ characters.

If count characters are available before an end of file, they are written into string starting at index start, and count is returned. If fewer characters are available before an end of file, but one or more can be read, those characters are written into string starting at index start and the number of characters actually read is returned as an exact integer object. If no characters can be read before an end of file, the end-of-file object is returned.

**get-string-all** input-port  
[Scheme Procedure]  
Reads from input-port until an end of file, decoding characters in the same manner as get-string-n and get-string-n!.

If characters are available before the end of file, a string containing all the characters decoded from that data are returned. If no character precedes the end of file, the end-of-file object is returned.

**get-line** input-port  
[Scheme Procedure]  
Reads from input-port up to and including the linefeed character or end of file, decoding characters in the same manner as get-string-n and get-string-n!.

If a linefeed character is read, a string containing all of the text up to (but not including) the linefeed character is returned, and the port is updated to point just past the linefeed character. If an end of file is encountered before any linefeed character is read, but some characters have been read and decoded as characters, a string containing those characters is returned. If an end of file is encountered before any characters are read, the end-of-file object is returned.

Finally, there are just two core procedures to write characters to a port.

**put-char** port char  
[Scheme Procedure]  
Writes char to the port. The put-char procedure returns an unspecified value.
put-string port string
put-string port string start
put-string port string start count

Write the count characters of string starting at index start to the port.

start and count must be non-negative exact integer objects. string must have a length of at least start + count. start defaults to 0. count defaults to (string - lengthstring) - start$.

Calling put-string is equivalent in all respects to calling put-char on the relevant sequence of characters, except that it will attempt to write multiple characters to the port at a time, even if the port is unbuffered.

The put-string procedure returns an unspecified value.

Textual ports have a textual position associated with them: a line and a column. Reading in characters or writing them out advances the line and the column appropriately.

port-column port
port-line port
scm_port_column (port)
scm_port_line (port)

Return the current column number or line number of port.

Port lines and positions are represented as 0-origin integers, which is to say that the first character of the first line is line 0, column 0. However, when you display a line number, for example in an error message, we recommend you add 1 to get 1-origin integers. This is because lines numbers traditionally start with 1, and that is what non-programmers will find most natural.

set-port-column! port column
set-port-line! port line
scm_set_port_column_x (port, column)
scm_set_port_line_x (port, line)

Set the current column or line number of port.

6.14.5 Simple Textual Output

Guile exports a simple formatted output function, simple-format. For a more capable formatted output facility, See Section 7.10 [Formatted Output], page 700.

simple-format destination message . args
scm_simple_format (destination, message, args)

Write message to destination, defaulting to the current output port. message can contain ~A and ~S escapes. When printed, the escapes are replaced with corresponding members of args: ~A formats using display and ~S formats using write. If destination is #t, then use the current output port, if destination is #f, then return a string containing the formatted text. Does not add a trailing newline.

Somewhat confusingly, Guile binds the format identifier to simple-format at startup. Once (ice-9 format) loads, it actually replaces the core format binding, so depending on whether you or a module you use has loaded (ice-9 format), you may be using the simple or the more capable version.
6.14.6 Buffering

Every port has associated input and output buffers. You can think of ports as being backed by some mutable store, and that store might be far away. For example, ports backed by file descriptors have to go all the way to the kernel to read and write their data. To avoid this round-trip cost, Guile usually reads in data from the mutable store in chunks, and then services small requests like get-char out of that intermediate buffer. Similarly, small writes like write-char first go to a buffer, and are sent to the store when the buffer is full (or when port is flushed). Buffered ports speed up your program by reducing the number of round-trips to the mutable store, and they do so in a way that is mostly transparent to the user.

There are two major ways, however, in which buffering affects program semantics. Building correct, performant programs requires understanding these situations.

The first case is in random-access read/write ports (see Section 6.14.7 [Random Access], page 337). These ports, usually backed by a file, logically operate over the same mutable store when both reading and writing. So, if you read a character, causing the buffer to fill, then write a character, the bytes you filled in your read buffer are now invalid. Every time you switch between reading and writing, Guile has to flush any pending buffer. If this happens frequently, the cost can be high. In that case you should reduce the amount that you buffer, in both directions. Similarly, Guile has to flush buffers before seeking. None of these considerations apply to sockets, which don’t logically read from and write to the same mutable store, and are not seekable. Note also that sockets are unbuffered by default. See Section 7.2.11.4 [Network Sockets and Communication], page 535.

The second case is the more pernicious one. If you write data to a buffered port, it probably doesn’t go out to the mutable store directly. (This “probably” introduces some indeterminism in your program: what goes to the store, and when, depends on how full the buffer is. It is something that the user needs to explicitly be aware of.) The data is written to the store later – when the buffer fills up due to another write, or when force-output is called, or when close-port is called, or when the program exits, or even when the garbage collector runs. The salient point is, the errors are signalled then too. Buffered writes defer error detection (and defer the side effects to the mutable store), perhaps indefinitely if the port type does not need to be closed at GC.

One common heuristic that works well for textual ports is to flush output when a newline (\n) is written. This line buffering mode is on by default for TTY ports. Most other ports are block buffered, meaning that once the output buffer reaches the block size, which depends on the port and its configuration, the output is flushed as a block, without regard to what is in the block. Likewise reads are read in at the block size, though if there are fewer bytes available to read, the buffer may not be entirely filled.

Note that binary reads or writes that are larger than the buffer size go directly to the mutable store without passing through the buffers. If your access pattern involves many big reads or writes, buffering might not matter so much to you.

To control the buffering behavior of a port, use setvbuf.

\begin{verbatim}
setvbuf port mode [size] [Scheme Procedure]
scm_setvbuf (port, mode, size) [C Function]
\end{verbatim}

Set the buffering mode for port. mode can be one of the following symbols:
null  non-buffered
line  line buffered
block block buffered, using a newly allocated buffer of size bytes. If size is
omitted, a default size will be used.

Another way to set the buffering, for file ports, is to open the file with 0 or 1 as part
of the mode string, for unbuffered or line-buffered ports, respectively. See Section 6.14.10.1
[File Ports], page 341, for more.

Any buffered output data will be written out when the port is closed. To make sure to
flush it at specific points in your program, use force-output.

force-output [port]  [Scheme Procedure]
scm_force_output (port)  [C Function]
Flush the specified output port, or the current output port if port is omitted. The
current output buffer contents, if any, are passed to the underlying port implementa-
tion.

The return value is unspecified.

flush-all-ports  [Scheme Procedure]
scm_flush_all_ports ()  [C Function]
Equivalent to calling force-output on all open output ports. The return value is
unspecified.

Similarly, sometimes you might want to switch from using Guile’s ports to working
directly on file descriptors. In that case, for input ports use drain-input to get any
buffered input from that port.

drain-input port  [Scheme Procedure]
scm_drain_input (port)  [C Function]
This procedure clears a port’s input buffers, similar to the way that force-output
clears the output buffer. The contents of the buffers are returned as a single string,
e.g.,
(define p (open-input-file ...))
(drain-input p) => empty string, nothing buffered yet.
(unread-char (read-char p) p)
(drain-input p) => initial chars from p, up to the buffer size.

All of these considerations are very similar to those of streams in the C library, although
Guile’s ports are not built on top of C streams. Still, it is useful to read what other systems
do. See Section “Streams” in The GNU C Library Reference Manual, for more discussion
on C streams.

6.14.7 Random Access

seek fd_port offset whence  [Scheme Procedure]
scm_seek (fd_port, offset, whence)  [C Function]
Sets the current position of fd_port to the integer offset. For a file port, offset is
expressed as a number of bytes; for other types of ports, such as string ports, offset
is an abstract representation of the position within the port’s data, not necessarily expressed as a number of bytes. offset is interpreted according to the value of whence. One of the following variables should be supplied for whence:

- **SEEK_SET**
  Seek from the beginning of the file.

- **SEEK_CUR**
  Seek from the current position.

- **SEEK_END**
  Seek from the end of the file.

If fd_port is a file descriptor, the underlying system call is lseek. port may be a string port.

The value returned is the new position in fd_port. This means that the current position of a port can be obtained using:

\[
\text{(seek port 0 SEEK_CUR)}
\]

**ftell** fd_port

*Scheme Procedure*

Return an integer representing the current position of fd_port, measured from the beginning. Equivalent to:

\[
\text{(seek port 0 SEEK_CUR)}
\]

**scm_ftell** (fd_port)

*C Function*

Truncate file to length bytes. file can be a filename string, a port object, or an integer file descriptor. The return value is unspecified.

For a port or file descriptor length can be omitted, in which case the file is truncated at the current position (per ftell above).

On most systems a file can be extended by giving a length greater than the current size, but this is not mandatory in the POSIX standard.

### 6.14.8 Line Oriented and Delimited Text

The delimited-I/O module can be accessed with:

\[
\text{(use-modules (ice-9 rdelim))}
\]

It can be used to read or write lines of text, or read text delimited by a specified set of characters.

**read-line** [port] [handle-delim]

*Scheme Procedure*

Return a line of text from port if specified, otherwise from the value returned by (current-input-port). Under Unix, a line of text is terminated by the first end-of-line character or by end-of-file.

If handle-delim is specified, it should be one of the following symbols:

- **trim** Discard the terminating delimiter. This is the default, but it will be impossible to tell whether the read terminated with a delimiter or end-of-file.
concat Append the terminating delimiter (if any) to the returned string.
peek Push the terminating delimiter (if any) back on to the port.
split Return a pair containing the string read from the port and the terminating delimiter or end-of-file object.

read-line! buf [port] [Scheme Procedure]
Read a line of text into the supplied string buf and return the number of characters added to buf. If buf is filled, then #f is returned. Read from port if specified, otherwise from the value returned by (current-input-port).

read-delimited delims [port] [handle-delim] [Scheme Procedure]
Read text until one of the characters in the string delims is found or end-of-file is reached. Read from port if supplied, otherwise from the value returned by (current-input-port). handle-delim takes the same values as described for read-line.

read-delimited! delims buf [port] [handle-delim] [start] [Scheme Procedure]
[endpoint]
Read text into the supplied string buf.
If a delimiter was found, return the number of characters written, except if handle-delim is split, in which case the return value is a pair, as noted above.
As a special case, if port was already at end-of-stream, the EOF object is returned. Also, if no characters were written because the buffer was full, #f is returned.
It’s something of a wacky interface, to be honest.

%read-delimited! delims str gobble [port [start [end]]] [Scheme Procedure]
scm_read_delimited_x (delims, str, gobble, port, start, end) [C Function]
Read characters from port into str until one of the characters in the delims string is encountered. If gobble is true, discard the delimiter character; otherwise, leave it in the input stream for the next read. If port is not specified, use the value of (current-input-port). If start or end are specified, store data only into the substring of str bounded by start and end (which default to the beginning and end of the string, respectively).
Return a pair consisting of the delimiter that terminated the string and the number of characters read. If reading stopped at the end of file, the delimiter returned is the eof-object; if the string was filled without encountering a delimiter, this value is #f.

%read-line [port] [Scheme Procedure]
scm_read_line (port) [C Function]
Read a newline-terminated line from port, allocating storage as necessary. The newline terminator (if any) is removed from the string, and a pair consisting of the line and its delimiter is returned. The delimiter may be either a newline or the eof-object; if %read-line is called at the end of file, it returns the pair (#<eof> . #<eof>).
6.14.9 Default Ports for Input, Output and Errors

**current-input-port**

Return the current input port. This is the default port used by many input procedures.

Initially this is the *standard input* in Unix and C terminology. When the standard input is a tty the port is unbuffered, otherwise it’s fully buffered.

Unbuffered input is good if an application runs an interactive subprocess, since any type-ahead input won’t go into Guile’s buffer and be unavailable to the subprocess.

Note that Guile buffering is completely separate from the tty “line discipline”. In the usual cooked mode on a tty Guile only sees a line of input once the user presses Return.

**current-output-port**

Return the current output port. This is the default port used by many output procedures.

Initially this is the *standard output* in Unix and C terminology. When the standard output is a tty this port is unbuffered, otherwise it’s fully buffered.

Unbuffered output to a tty is good for ensuring progress output or a prompt is seen. But an application which always prints whole lines could change to line buffered, or an application with a lot of output could go fully buffered and perhaps make explicit force-output calls (see Section 6.14.6 [Buffering], page 336) at selected points.

**current-error-port**

Return the port to which errors and warnings should be sent.

Initially this is the *standard error* in Unix and C terminology. When the standard error is a tty this port is unbuffered, otherwise it’s fully buffered.

**set-current-input-port**

Change the ports returned by current-input-port, current-output-port and current-error-port, respectively, so that they use the supplied port for input or output.

**set-current-output-port**

**set-current-error-port**

**with-input-from-port**

**with-output-to-port**

**with-error-to-port**

Call thunk in a dynamic environment in which current-input-port, current-output-port or current-error-port is rebound to the given port.
Chapter 6: API Reference

void scm_dynwind_current_input_port (SCM port)  [C Function]
void scm_dynwind_current_output_port (SCM port)  [C Function]
void scm_dynwind_current_error_port (SCM port)  [C Function]

These functions must be used inside a pair of calls to scm_dynwind_begin and scm_dynwind_end (see Section 6.13.10 [Dynamic Wind], page 317). During the dynwind context, the indicated port is set to port.

More precisely, the current port is swapped with a 'backup' value whenever the dynwind context is entered or left. The backup value is initialized with the port argument.

6.14.10 Types of Port

6.14.10.1 File Ports

The following procedures are used to open file ports. See also Section 7.2.2 [Ports and File Descriptors], page 493, for an interface to the Unix open system call.

All file access uses the “LFS” large file support functions when available, so files bigger than 2 Gbytes (2^31 bytes) can be read and written on a 32-bit system.

Most systems have limits on how many files can be open, so it’s strongly recommended that file ports be closed explicitly when no longer required (see Section 6.14.1 [Ports], page 328).

open-file filename mode [#:guess-encoding=#f]  [Scheme Procedure]
  [#:encoding=#f]
scm_open_file_with_encoding (filename, mode, guess_encoding, encoding)  [C Function]
scm_open_file (filename, mode)  [C Function]

Open the file whose name is filename, and return a port representing that file. The attributes of the port are determined by the mode string. The way in which this is interpreted is similar to C stdio. The first character must be one of the following:

‘r’ Open an existing file for input.

‘w’ Open a file for output, creating it if it doesn’t already exist or removing its contents if it does.

‘a’ Open a file for output, creating it if it doesn’t already exist. All writes to the port will go to the end of the file. The "append mode" can be turned off while the port is in use see Section 7.2.2 [Ports and File Descriptors], page 493,

The following additional characters can be appended:

‘+’ Open the port for both input and output. E.g., r+: open an existing file for both input and output.

‘0’ Create an "unbuffered" port. In this case input and output operations are passed directly to the underlying port implementation without additional buffering. This is likely to slow down I/O operations. The buffering mode can be changed while a port is in use (see Section 6.14.6 [Buffering], page 336).
`l` Add line-buffering to the port. The port output buffer will be automatically flushed whenever a newline character is written.

`b` Use binary mode, ensuring that each byte in the file will be read as one Scheme character.

To provide this property, the file will be opened with the 8-bit character encoding "ISO-8859-1", ignoring the default port encoding. See Section 6.14.1 [Ports], page 328, for more information on port encodings.

Note that while it is possible to read and write binary data as characters or strings, it is usually better to treat bytes as octets, and byte sequences as bytevectors. See Section 6.14.2 [Binary I/O], page 330, for more.

This option had another historical meaning, for DOS compatibility: in the default (textual) mode, DOS reads a CR-LF sequence as one LF byte. The `b` flag prevents this from happening, adding `O_BINARY` to the underlying `open` call. Still, the flag is generally useful because of its port encoding ramifications.

Unless binary mode is requested, the character encoding of the new port is determined as follows: First, if `guess-encoding` is true, the `file-encoding` procedure is used to guess the encoding of the file (see Section 6.18.8 [Character Encoding of Source Files], page 392). If `guess-encoding` is false or if `file-encoding` fails, `encoding` is used unless it is also false. As a last resort, the default port encoding is used. See Section 6.14.1 [Ports], page 328, for more information on port encodings. It is an error to pass a non-true `guess-encoding` or `encoding` if binary mode is requested.

If a file cannot be opened with the access requested, `open-file` throws an exception.

```scheme
(open-file filename
  (if binary "rb" "r")
  #:guess-encoding guess-encoding
  #:encoding encoding)
```

```scheme
(open-file filename
  (if binary "wb" "w")
  #:encoding encoding)
```
call-with-input-file  filename proc  [Scheme Procedure]
[#:guess-encoding=#f] [#:encoding=#f] [#:binary=#f]
call-with-output-file filename proc [#:encoding=#f]  [Scheme Procedure]
[#:binary=#f]
Open filename for input or output, and call (proc port) with the resulting port. Return the value returned by proc. filename is opened as per open-input-file or open-output-file respectively, and an error is signaled if it cannot be opened.
When proc returns, the port is closed. If proc does not return (e.g. if it throws an error), then the port might not be closed automatically, though it will be garbage collected in the usual way if not otherwise referenced.

with-input-from-file filename thunk  [Scheme Procedure]
[#:guess-encoding=#f] [#:encoding=#f] [#:binary=#f]
with-output-to-file filename thunk [#:encoding=#f]  [Scheme Procedure]
[#:binary=#f]
with-error-to-file filename thunk [#:encoding=#f]  [Scheme Procedure]
[#:binary=#f]
Open filename and call (thunk) with the new port setup as respectively the current-input-port, current-output-port, or current-error-port. Return the value returned by thunk. filename is opened as per open-input-file or open-output-file respectively, and an error is signaled if it cannot be opened.
When thunk returns, the port is closed and the previous setting of the respective current port is restored.
The current port setting is managed with dynamic-wind, so the previous value is restored no matter how thunk exits (e.g. an exception), and if thunk is re-entered (via a captured continuation) then it’s set again to the filename port.
The port is closed when thunk returns normally, but not when exited via an exception or new continuation. This ensures it’s still ready for use if thunk is re-entered by a captured continuation. Of course the port is always garbage collected and closed in the usual way when no longer referenced anywhere.

port-mode port  [Scheme Procedure]
scm_port_mode (port)  [C Function]
Return the port modes associated with the open port port. These will not necessarily be identical to the modes used when the port was opened, since modes such as "append" which are used only during port creation are not retained.

port-filename port  [Scheme Procedure]
scm_port_filename (port)  [C Function]
Return the filename associated with port, or #f if no filename is associated with the port.
port must be open; port-filename cannot be used once the port is closed.

set-port-filename! port filename  [Scheme Procedure]
scm_set_port_filename_x (port, filename)  [C Function]
Change the filename associated with port, using the current input port if none is specified. Note that this does not change the port’s source of data, but only the value that is returned by port-filename and reported in diagnostic output.
file-port? obj

scm_file_port_p (obj)

Determine whether obj is a port that is related to a file.

6.14.10.2 Bytevector Ports

open-bytevector-input-port bv [transcoder]

scm_open_bytevector_input_port (bv, transcoder)

Return an input port whose contents are drawn from bytevector bv (see Section 6.6.12 [Bytevectors], page 193).

The transcoder argument is currently not supported.

open-bytevector-output-port [transcoder]

scm_open_bytevector_output_port (transcoder)

Return two values: a binary output port and a procedure. The latter should be called with zero arguments to obtain a bytevector containing the data accumulated by the port, as illustrated below.

\[
\begin{align*}
\text{(call-with-values} & \text{ (lambda () )} \\
& \text{(open-bytevector-output-port())} \\
& \text{(lambda (port get-bytevector))} \\
& \text{(display "hello" port)} \\
& \text{(get-bytevector()))}
\end{align*}
\]

⇒ #vu8(104 101 108 108 111)

The transcoder argument is currently not supported.

6.14.10.3 String Ports

call-with-output-string proc

scm_call_with_output_string (proc)

Calls the one-argument procedure proc with a newly created output port. When the function returns, the string composed of the characters written into the port is returned. proc should not close the port.

call-with-input-string string proc

scm_call_with_input_string (string, proc)

Calls the one-argument procedure proc with a newly created input port from which string’s contents may be read. The value yielded by the proc is returned.

with-output-to-string thunk

Calls the zero-argument procedure thunk with the current output port set temporarily to a new string port. It returns a string composed of the characters written to the current output.

with-input-from-string string thunk

Calls the zero-argument procedure thunk with the current input port set temporarily to a string port opened on the specified string. The value yielded by thunk is returned.
open-input-string \texttt{str} \hspace{1cm} \textbf{[Scheme Procedure]}  
\texttt{scm_open_input_string (str)} \hspace{1cm} \textbf{[C Function]}  
Take a string and return an input port that delivers characters from the string. The port can be closed by \texttt{close-input-port}, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

open-output-string \hspace{1cm} \textbf{[Scheme Procedure]}  
\texttt{scm_open_output_string ()} \hspace{1cm} \textbf{[C Function]}  
Return an output port that will accumulate characters for retrieval by \texttt{get-output-string}. The port can be closed by the procedure \texttt{close-output-port}, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

get-output-string \texttt{port} \hspace{1cm} \textbf{[Scheme Procedure]}  
\texttt{scm_get_output_string (port)} \hspace{1cm} \textbf{[C Function]}  
Given an output port created by \texttt{open-output-string}, return a string consisting of the characters that have been output to the port so far. 
\texttt{get-output-string} must be used before closing \texttt{port}, once closed the string cannot be obtained.

With string ports, the port-encoding is treated differently than other types of ports. When string ports are created, they do not inherit a character encoding from the current locale. They are given a default locale that allows them to handle all valid string characters. Typically one should not modify a string port’s character encoding away from its default. See Section 6.14.3 [Encoding], page 331.

### 6.14.10.4 Custom Ports

Custom ports allow the user to provide input and handle output via user-supplied procedures. Guile currently only provides custom binary ports, not textual ports; for custom textual ports, See Section 6.14.10.5 [Soft Ports], page 347. We should add the R6RS custom textual port interfaces though. Contributions are appreciated.

\texttt{make-custom-binary-input-port id read! get-position} \hspace{1cm} \textbf{[Scheme Procedure]}  
\texttt{set-position! close} \hspace{1cm} \textbf{}  
Return a new custom binary input port\(^8\) named \texttt{id} (a string) whose input is drained by invoking \texttt{read!} and passing it a bytevector, an index where bytes should be written, and the number of bytes to read. The \texttt{read!} procedure must return an integer indicating the number of bytes read, or 0 to indicate the end-of-file.

Optionally, if \texttt{get-position} is not \texttt{#f}, it must be a thunk that will be called when \texttt{port-position} is invoked on the custom binary port and should return an integer indicating the position within the underlying data stream; if \texttt{get-position} was not supplied, the returned port does not support \texttt{port-position}.

Likewise, if \texttt{set-position!} is not \texttt{#f}, it should be a one-argument procedure. When \texttt{set-port-position!} is invoked on the custom binary input port, \texttt{set-position!} is passed an integer indicating the position of the next byte is to read.

Finally, if \texttt{close} is not \texttt{#f}, it must be a thunk. It is invoked when the custom binary input port is closed.

\(^8\) This is similar in spirit to Guile’s soft ports (see Section 6.14.10.5 [Soft Ports], page 347).
The returned port is fully buffered by default, but its buffering mode can be changed using \texttt{setvbuf} (see Section 6.14.6 \[Buffering\], page 336).

Using a custom binary input port, the \texttt{open-bytevector-input-port} procedure (see Section 6.14.10.2 \[Bytevector Ports\], page 344) could be implemented as follows:

\begin{verbatim}
(define (open-bytevector-input-port source)
  (define position 0)
  (define length (bytevector-length source))

  (define (read! bv start count)
    (let ((count (min count (- length position))))
      (bytevector-copy! source position
        bv start count)
      (set! position (+ position count))
      count))

  (define (get-position) position)

  (define (set-position! new-position)
    (set! position new-position))

  (make-custom-binary-input-port "the port" read! get-position set-position!
    #f))

(read (open-bytevector-input-port (string->utf8 "hello")))
⇒ hello
\end{verbatim}

\textbf{make-custom-binary-output-port} $\text{id \ write! \ get-position \ close}$ \hspace{1cm} \text{[Scheme Procedure]}

Return a new custom binary output port named \text{id} (a string) whose output is sunk by invoking \texttt{write!} and passing it a bytevector, an index where bytes should be read from this bytevector, and the number of bytes to be “written”. The \texttt{write!} procedure must return an integer indicating the number of bytes actually written; when it is passed 0 as the number of bytes to write, it should behave as though an end-of-file was sent to the byte sink.

The other arguments are as for \texttt{make-custom-binary-input-port}.

\textbf{make-custom-binary-input/output-port} $\text{id \ read! \ write! \ get-position \ set-position! \ close}$ \hspace{1cm} \text{[Scheme Procedure]}

Return a new custom binary input/output port named \text{id} (a string). The various arguments are the same as for \texttt{make-custom-binary-input-port} and \texttt{make-custom-binary-output-port}. If buffering is enabled on the port, as is the case by default, input will be buffered in both directions; See Section 6.14.6 \[Buffering\], page 336. If the \texttt{set-position!} function is provided and not \texttt{#f}, then the port will also be marked as random-access, causing the buffer to be flushed between reads and writes.
6.14.10.5 Soft Ports

A soft port is a port based on a vector of procedures capable of accepting or delivering characters. It allows emulation of I/O ports.

**make-soft-port** \(pv\) **modes**

[Scheme Procedure]

Return a port capable of receiving or delivering characters as specified by the **modes** string (see Section 6.14.10.1 [File Ports], page 341). \(pv\) must be a vector of length 5 or 6. Its components are as follows:

0. procedure accepting one character for output
1. procedure accepting a string for output
2. thunk for flushing output
3. thunk for getting one character
4. thunk for closing port (not by garbage collection)
5. (if present and not \#f) thunk for computing the number of characters that can be read from the port without blocking.

For an output-only port only elements 0, 1, 2, and 4 need be procedures. For an input-only port only elements 3 and 4 need be procedures. Thunks 2 and 4 can instead be \#f if there is no useful operation for them to perform.

If thunk 3 returns \#f or an **eof-object** (see Section “Input” in *The Revised^5 Report on Scheme*) it indicates that the port has reached end-of-file. For example:

```scheme
(define stdout (current-output-port))
(define p (make-soft-port
  (vector
    (lambda (c) (write c stdout))
    (lambda (s) (display s stdout))
    (lambda () (display "." stdout))
    (lambda () (char-upcase (read-char)))
    (lambda () (display "@" stdout)))
	"rw"))

(write p p) ⇒ #<input-output: soft 8081e20>
```

6.14.10.6 Void Ports

This kind of port causes any data to be discarded when written to, and always returns the end-of-file object when read from.

%**make-void-port** **mode**

[Scheme Procedure]

Create and return a new void port. A void port acts like /dev/null. The **mode** argument specifies the input/output modes for this port: see the documentation for **open-file** in Section 6.14.10.1 [File Ports], page 341.

6.14.11 Venerable Port Interfaces

Over the 25 years or so that Guile has been around, its port system has evolved, adding many useful features. At the same time there have been four major Scheme standards
released in those 25 years, which also evolve the common Scheme understanding of what a port interface should be. Alas, it would be too much to ask for all of these evolutionary branches to be consistent. Some of Guile’s original interfaces don’t mesh with the later Scheme standards, and yet Guile can’t just drop old interfaces. Sadly as well, the R6RS and R7RS standards both part from a base of R5RS, but end up in different and somewhat incompatible designs.

Guile’s approach is to pick a set of port primitives that make sense together. We document that set of primitives, design our internal interfaces around them, and recommend them to users. As the R6RS I/O system is the most capable standard that Scheme has yet produced in this domain, we mostly recommend that; (ice-9 binary-ports) and (ice-9 textual-ports) are wholly modelled on (rnrs io ports). Guile does not wholly copy R6RS, however; See Section 7.6.1 [R6RS Incompatibilities], page 648.

At the same time, we have many venerable port interfaces, lore handed down to us from our hacker ancestors. Most of these interfaces even predate the expectation that Scheme should have modules, so they are present in the default environment. In Guile we support them as well and we have no plans to remove them, but again we don’t recommend them for new users.

\[\text{char-ready? \;} [\text{port}]\]  
[Scheme Procedure]  
Return \#t if a character is ready on input \text{port} and return \#f otherwise. If \text{char-ready?} returns \#t then the next \text{read-char} operation on \text{port} is guaranteed not to hang. If \text{port} is a file port at end of file then \text{char-ready?} returns \#t.

\text{char-ready?} exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must make sure that characters whose existence has been asserted by \text{char-ready?} cannot be rubbed out. If \text{char-ready?} were to return \#f at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

Note that \text{char-ready?} only works reliably for terminals and sockets with one-byte encodings. Under the hood it will return \#t if the port has any input buffered, or if the file descriptor that backs the port polls as readable, indicating that Guile can fetch more bytes from the kernel. However being able to fetch one byte doesn’t mean that a full character is available; See Section 6.14.3 [Encoding], page 331. Also, on many systems it’s possible for a file descriptor to poll as readable, but then block when it comes time to read bytes. Note also that on Linux kernels, all file ports backed by files always poll as readable. For non-file ports, this procedure always returns \#t, except for soft ports, which have a \text{char-ready?} handler. See Section 6.14.10.5 [Soft Ports], page 347.

In short, this is a legacy procedure whose semantics are hard to provide. However it is a useful check to see if any input is buffered. See Section 6.14.14 [Non-Blocking I/O], page 353.

\[\text{read-char \;} [\text{port}]\]  
[Scheme Procedure]  
The same as \text{get-char}, except that \text{port} defaults to the current input port. See Section 6.14.4 [Textual I/O], page 333.
peek-char [port]  
[Scheme Procedure]
The same as `lookahead-char`, except that `port` defaults to the current input port. See Section 6.14.4 [Textual I/O], page 333.

unread-char cobj [port]  
[Scheme Procedure]
The same as `unget-char`, except that `port` defaults to the current input port, and the arguments are swapped. See Section 6.14.4 [Textual I/O], page 333.

unread-string str port  
[Scheme Procedure]
scm_unread_string (str, port)  
[C Function]
The same as `unget-string`, except that `port` defaults to the current input port, and the arguments are swapped. See Section 6.14.4 [Textual I/O], page 333.

newline [port]  
[Scheme Procedure]
Send a newline to `port`. If `port` is omitted, send to the current output port. Equivalent to `(put-char port #\newline)`.

write-char chr [port]  
[Scheme Procedure]
The same as `put-char`, except that `port` defaults to the current input port, and the arguments are swapped. See Section 6.14.4 [Textual I/O], page 333.

6.14.12 Using Ports from C

Guile’s C interfaces provides some niceties for sending and receiving bytes and characters in a way that works better with C.

size_t scm_c_read (SCM port, void *buffer, size_t size)  
[C Function]
Read up to `size` bytes from `port` and store them in `buffer`. The return value is the number of bytes actually read, which can be less than `size` if end-of-file has been reached.

Note that as this is a binary input procedure, this function does not update `port-line` and `port-column` (see Section 6.14.4 [Textual I/O], page 333).

void scm_c_write (SCM port, const void *buffer, size_t size)  
[C Function]
Write `size` bytes at `buffer` to `port`.

Note that as this is a binary output procedure, this function does not update `port-line` and `port-column` (see Section 6.14.4 [Textual I/O], page 333).

size_t scm_c_read_bytes (SCM port, SCM bv, size_t start, size_t count)  
[C Function]
void scm_c_write_bytes (SCM port, SCM bv, size_t start, size_t count)  
[C Function]
Like `scm_c_read` and `scm_c_write`, but reading into or writing from the bytevector `bv`. `count` indicates the byte index at which to start in the bytevector, and the read or write will continue for `count` bytes.

void scm_unget_bytes (const unsigned char *buf, size_t len, SCM port)  
[C Function]
void scm_unget_byte (int c, SCM port)  
[C Function]
void scm_ungetc (scm_t wchar c, SCM port)  
[C Function]
Like `unget-bytevector`, `unget-byte`, and `unget-char`, respectively. See Section 6.14.4 [Textual I/O], page 333.
6.14.13 Implementing New Port Types in C

This section describes how to implement a new port type in C. Although ports support many operations, as a data structure they present an opaque interface to the user. To the port implementor, you have two pieces of information to work with: the port type, and the port’s “stream”. The port type is an opaque pointer allocated when defining your port type. It is your key into the port API, and it helps you identify which ports are actually yours. The “stream” is a pointer you control, and which you set when you create a port. Get a stream from a port using the SCM_STREAM macro. Note that your port methods are only ever called with ports of your type.

A port type is created by calling scm_make_port_type. Once you have your port type, you can create ports with scm_c_make_port, or scm_c_make_port_with_encoding.

The mode bits should include one or more of the flags SCM_RDNG or SCM_WRTNG, indicating that the port is an input and/or an output port, respectively. The mode bits may also include SCM_BUF0 or SCM_BUFLINE, indicating that the port should be unbuffered or line-buffered, respectively. The default is that the port will be block-buffered. See Section 6.14.6 [Buffering], page 336.

As you would imagine, encoding and conversion_strategy specify the port’s initial textual encoding and conversion strategy. Both are symbols. scm_c_make_port is the same as scm_c_make_port_with_encoding, except it uses the default port encoding and conversion strategy.

The port type has a number of associate procedures and properties which collectively implement the port’s behavior. Creating a new port type mostly involves writing these procedures.
name
A pointer to a NUL terminated string: the name of the port type. This property
is initialized via the first argument to scm_make_port_type.

read
A port’s read implementation fills read buffers. It should copy bytes to the
supplied bytevector dst, starting at offset start and continuing for count bytes,
returning the number of bytes read.

write
A port’s write implementation flushes write buffers to the mutable store. A
port’s read implementation fills read buffers. It should write out bytes from
the supplied bytevector src, starting at offset start and continuing for count
bytes, and return the number of bytes that were written.

read_wait_fd
write_wait_fd
If a port’s read or write function returns (size_t) -1, that indicates that
reading or writing would block. In that case to preserve the illusion of a blocking
read or write operation, Guile’s C port run-time will poll on the file descriptor
returned by either the port’s read_wait_fd or write_wait_fd function. Set
using

void scm_set_port_read_wait_fd (scm.t.port.type *type, int (*wait_fd) (SCM port))
void scm_set_port_write_wait_fd (scm.t.port.type *type, int (*wait_fd) (SCM port))

Only a port type which implements the read_wait_fd or write_wait_fd port
methods can usefully return (size_t) -1 from a read or write function. See
Section 6.14.14 [Non-Blocking I/O], page 353, for more on non-blocking I/O in
Guile.

print
Called when write is called on the port, to print a port description. For
example, for a file port it may produce something like: #<input: /etc/passwd 3>. Set using

void scm_set_port_print (scm.t.port.type *type, int (*print) (SCM port, SCM dest_port, scm.print.state *pstate))

The first argument port is the port being printed, the second argument
dest_port is where its description should go.

close
Called when the port is closed. It should free any resources used by the port.
Set using

void scm_set_port_close (scm.t.port.type *type, void (*close) (SCM port))

By default, ports that are garbage collected just go away without closing. If
your port type needs to release some external resource like a file descriptor,
or needs to make sure that its internal buffers are flushed even if the port is
collected while it was open, then mark the port type as needing a close on GC.

void scm_set_port_needs_close_on_gc (scm.t.port.type *type, int needs_close_p)

seek
Set the current position of the port. Guile will flush read and/or write buffers
before seeking, as appropriate.
**scm_set_port_seek** (scm_t_port_type *type, scm_t_off (*seek) (SCM port, scm_t_off offset, int whence))

*truncate*  
Truncate the port data to be specified length. Guile will flush buffers before hand, as appropriate. Set using

**scm_set_port_truncate** (scm_t_port_type *type, void (*truncate) (SCM port, scm_t_off length))

*random_access_p*  
Determine whether this port is a random-access port.

Seeking on a random-access port with buffered input, or switching to writing after reading, will cause the buffered input to be discarded and Guile will seek the port back the buffered number of bytes. Likewise seeking on a random-access port with buffered output, or switching to reading after writing, will flush pending bytes with a call to the *write* procedure. See Section 6.14.6 [Buffering], page 336.

Indicate to Guile that your port needs this behavior by returning a nonzero value from your *random_access_p* function. The default implementation of this function returns nonzero if the port type supplies a seek implementation.

**scm_set_port_random_access_p** (scm_t_port_type *type, int (*random_access_p) (SCM port));

*get_natural_buffer_sizes*  
Guile will internally attach buffers to ports. An input port always has a read buffer and an output port always has a write buffer. See Section 6.14.6 [Buffering], page 336. A port buffer consists of a bytevector, along with some cursors into that bytevector denoting where to get and put data.

Port implementations generally don’t have to be concerned with buffering: a port type’s *read* or *write* function will receive the buffer’s bytevector as an argument, along with an offset and a length into that bytevector, and should then either fill or empty that bytevector. However in some cases, port implementations may be able to provide an appropriate default buffer size to Guile.

**scm_set_port_get_natural_buffer_sizes** (scm_t_port_type *type, void (*get_natural_buffer_sizes) (SCM, size_t *read_buf_size, size_t *write_buf_size))

Fill in *read_buf_size* and *write_buf_size* with an appropriate buffer size for this port, if one is known.

File ports implement a *get_natural_buffer_sizes* to let the operating system inform Guile about the appropriate buffer sizes for the particular file opened by the port.

Note that calls to all of these methods can proceed in parallel and concurrently and from any thread up until the point that the port is closed. The call to *close* will happen when no other method is running, and no method will be called after the *close* method is called. If your port implementation needs mutual exclusion to prevent concurrency, it is responsible for locking appropriately.
6.14.14 Non-Blocking I/O

Most ports in Guile are blocking: when you try to read a character from a port, Guile will block on the read until a character is ready, or end-of-stream is detected. Likewise whenever Guile goes to write (possibly buffered) data to an output port, Guile will block until all the data is written.

Interacting with ports in blocking mode is very convenient: you can write straightforward, sequential algorithms whose code flow reflects the flow of data. However, blocking I/O has two main limitations.

The first is that it’s easy to get into a situation where code is waiting on data. Time spent waiting on data when code could be doing something else is wasteful and prevents your program from reaching its peak throughput. If you implement a web server that sequentially handles requests from clients, it’s very easy for the server to end up waiting on a client to finish its HTTP request, or waiting on it to consume the response. The end result is that you are able to serve fewer requests per second than you’d like to serve.

The second limitation is related: a blocking parser over user-controlled input is a denial-of-service vulnerability. Indeed the so-called “slow loris” attack of the early 2010s was just that: an attack on common web servers that drip-fed HTTP requests, one character at a time. All it took was a handful of slow loris connections to occupy an entire web server.

In Guile we would like to preserve the ability to write straightforward blocking networking processes of all kinds, but under the hood to allow those processes to suspend their requests if they would block.

To do this, the first piece is to allow Guile ports to declare themselves as being nonblocking. This is currently supported only for file ports, which also includes sockets, terminals, or any other port that is backed by a file descriptor. To do that, we use an arcane UNIX incantation:

```lisp
(let ((flags (fcntl socket F_GETFL)))
  (fcntl socket F_SETFL (logior O_NONBLOCK flags)))
```

Now the file descriptor is open in non-blocking mode. If Guile tries to read or write from this file and the read or write returns a result indicating that more data can only be had by doing a blocking read or write, Guile will block by polling on the socket’s `read-wait-fd` or `write-wait-fd`, to preserve the illusion of a blocking read or write. See Section 6.14.13 [I/O Extensions], page 350, for more on those internal interfaces.

So far we have just reproduced the status quo: the file descriptor is non-blocking, but the operations on the port do block. To go farther, it would be nice if we could suspend the “thread” using delimited continuations, and only resume the thread once the file descriptor is readable or writable. (See Section 6.13.5 [Prompts], page 303).

But here we run into a difficulty. The ports code is implemented in C, which means that although we can suspend the computation to some outer prompt, we can’t resume it because Guile can’t resume delimited continuations that capture the C stack.

To overcome this difficulty we have created a compatible but entirely parallel implementation of port operations. To use this implementation, do the following:

```lisp
(use-modules (ice-9 suspendable-ports))
(install-suspendable-ports!)
```
This will replace the core I/O primitives like `get-char` and `put-bytevector` with new versions that are exactly the same as the ones in the standard library, but with two differences. One is that when a read or a write would block, the suspendable port operations call out the value of the `current-read-waiter` or `current-write-waiter` parameter, as appropriate. See Section 6.13.12 [Parameters], page 324. The default read and write waiters do the same thing that the C read and write waiters do, which is to poll. User code can parameterize the waiters, though, enabling the computation to suspend and allow the program to process other I/O operations. Because the new suspendable ports implementation is written in Scheme, that suspended computation can resume again later when it is able to make progress. Success!

The other main difference is that because the new ports implementation is written in Scheme, it is slower than C, currently by a factor of 3 or 4, though it depends on many factors. For this reason we have to keep the C implementations as the default ones. One day when Guile’s compiler is better, we can close this gap and have only one port operation implementation again.

Note that Guile does not currently include an implementation of the facility to suspend the current thread and schedule other threads in the meantime. Before adding such a thing, we want to make sure that we’re providing the right primitives that can be used to build schedulers and other user-space concurrency patterns, and that the patterns that we settle on are the right patterns. In the meantime, have a look at 8sync (https://gnu.org/software/8sync) for a prototype of an asynchronous I/O and concurrency facility.

```
install-suspendable-ports!
```

Replace the core ports implementation with suspendable ports, as described above. This will mutate the values of the bindings like `get-char`, `put-u8`, and so on in place.

```
uninstall-suspendable-ports!
```

Restore the original core ports implementation, un-doing the effect of `install-suspendable-ports!`.

```
current-read-waiter
```

```
current-write-waiter
```

Parameters whose values are procedures of one argument, called when a suspendable port operation would block on a port while reading or writing, respectively. The default values of these parameters do a blocking `poll` on the port’s file descriptor. The procedures are passed the port in question as their one argument.

6.14.15 Handling of Unicode Byte Order Marks

This section documents the finer points of Guile’s handling of Unicode byte order marks (BOMs). A byte order mark (U+FEFF) is typically found at the start of a UTF-16 or UTF-32 stream, to allow readers to reliably determine the byte order. Occasionally, a BOM is found at the start of a UTF-8 stream, but this is much less common and not generally recommended.

Guile attempts to handle BOMs automatically, and in accordance with the recommendations of the Unicode Standard, when the port encoding is set to UTF-8, UTF-16, or UTF-32. In brief, Guile automatically writes a BOM at the start of a UTF-16 or UTF-32 stream, and automatically consumes one from the start of a UTF-8, UTF-16, or UTF-32 stream.
As specified in the Unicode Standard, a BOM is only handled specially at the start of a stream, and only if the port encoding is set to UTF-8, UTF-16 or UTF-32. If the port encoding is set to UTF-16BE, UTF-16LE, UTF-32BE, or UTF-32LE, then BOMs are not handled specially, and none of the special handling described in this section applies.

- To ensure that Guile will properly detect the byte order of a UTF-16 or UTF-32 stream, you must perform a textual read before any writes, seeks, or binary I/O. Guile will not attempt to read a BOM unless a read is explicitly requested at the start of the stream.
- If a textual write is performed before the first read, then an arbitrary byte order will be chosen. Currently, big endian is the default on all platforms, but that may change in the future. If you wish to explicitly control the byte order of an output stream, set the port encoding to UTF-16BE, UTF-16LE, UTF-32BE, or UTF-32LE, and explicitly write a BOM (\U0000Doc) if desired.
- If set-port-encoding! is called in the middle of a stream, Guile treats this as a new logical “start of stream” for purposes of BOM handling, and will forget about any BOMs that had previously been seen. Therefore, it may choose a different byte order than had been used previously. This is intended to support multiple logical text streams embedded within a larger binary stream.
- Binary I/O operations are not guaranteed to update Guile’s notion of whether the port is at the “start of the stream”, nor are they guaranteed to produce or consume BOMs.
- For ports that support seeking (e.g. normal files), the input and output streams are considered linked: if the user reads first, then a BOM will be consumed (if appropriate), but later writes will not produce a BOM. Similarly, if the user writes first, then later reads will not consume a BOM.
- For ports that are not random access (e.g. pipes, sockets, and terminals), the input and output streams are considered independent for purposes of BOM handling: the first read will consume a BOM (if appropriate), and the first write will also produce a BOM (if appropriate). However, the input and output streams will always use the same byte order.
- Seeks to the beginning of a file will set the “start of stream” flags. Therefore, a subsequent textual read or write will consume or produce a BOM. However, unlike set-port-encoding!, if a byte order had already been chosen for the port, it will remain in effect after a seek, and cannot be changed by the presence of a BOM. Seeks anywhere other than the beginning of a file clear the “start of stream” flags.

6.15 Regular Expressions

A regular expression (or regexp) is a pattern that describes a whole class of strings. A full description of regular expressions and their syntax is beyond the scope of this manual.

If your system does not include a POSIX regular expression library, and you have not linked Guile with a third-party regexp library such as Rx, these functions will not be available. You can tell whether your Guile installation includes regular expression support by checking whether (provided? 'regex) returns true.

The following regexp and string matching features are provided by the (ice-9 regex) module. Before using the described functions, you should load this module by executing (use-modules (ice-9 regex)).
6.15.1 Regexp Functions

By default, Guile supports POSIX extended regular expressions. That means that the characters ‘(’, ‘)’, ‘+’ and ‘?’ are special, and must be escaped if you wish to match the literal characters and there is no support for “non-greedy” variants of ‘*’, ‘+’ or ‘?’. This regular expression interface was modeled after that implemented by SCSH, the Scheme Shell. It is intended to be upwardly compatible with SCSH regular expressions.

Zero bytes (#\null) cannot be used in regex patterns or input strings, since the underlying C functions treat that as the end of string. If there’s a zero byte an error is thrown.

Internally, patterns and input strings are converted to the current locale’s encoding, and then passed to the C library’s regular expression routines (see Section “Regular Expressions” in The GNU C Library Reference Manual). The returned match structures always point to characters in the strings, not to individual bytes, even in the case of multi-byte encodings.

```
(string-match "[0-9][0-9][0-9][0-9]" "blah2002")
⇒ #(#"blah2002" (4 . 8))

(string-match "[A-Za-z]" "123456")
⇒ #f
```

Each time string-match is called, it must compile its pattern argument into a regular expression structure. This operation is expensive, which makes string-match inefficient if the same regular expression is used several times (for example, in a loop). For better performance, you can compile a regular expression in advance and then match strings against the compiled regexp.

```
make-regexp pat flag . . .
```

```
scm_make_regexp (pat, flaglst)
```

Compile the regular expression described by pat, and return the compiled regexp structure. If pat does not describe a legal regular expression, make-regexp throws a regular-expression-syntax error.

The flag arguments change the behavior of the compiled regular expression. The following values may be supplied:

```
regexp/icase
```

Consider uppercase and lowercase letters to be the same when matching.
regexp/newline  [Variable]
If a newline appears in the target string, then permit the ‘^’ and ‘$’ operators to match immediately after or immediately before the newline, respectively. Also, the ‘.’ and ‘[^...]’ operators will never match a newline character. The intent of this flag is to treat the target string as a buffer containing many lines of text, and the regular expression as a pattern that may match a single one of those lines.

regexp/basic  [Variable]
Compile a basic (“obsolete”) regexp instead of the extended (“modern”) regexps that are the default. Basic regexps do not consider ‘|’, ‘+’ or ‘?’ to be special characters, and require the ‘(...)’ and ‘(…)’ metacharacters to be backslash-escaped (see Section 6.15.3 [Backslash Escapes], page 361). There are several other differences between basic and extended regular expressions, but these are the most significant.

regexp/extended  [Variable]
Compile an extended regular expression rather than a basic regexp. This is the default behavior; this flag will not usually be needed. If a call to make-regexp includes both regexp/basic and regexp/extended flags, the one which comes last will override the earlier one.

regexp-exec  [Scheme Procedure]
scm_regexp_exec (rx, str, start, flags)

Match the compiled regular expression rx against str. If the optional integer start argument is provided, begin matching from that position in the string. Return a match structure describing the results of the match, or #f if no match could be found.

The flags argument changes the matching behavior. The following flag values may be supplied, use logior (see Section 6.6.2.13 [Bitwise Operations], page 125) to combine them,

regexp/notbol  [Variable]
Consider that the start offset into str is not the beginning of a line and should not match operator ‘^’.

If rx was created with the regexp/newline option above, ‘^’ will still match after a newline in str.

regexp/noteol  [Variable]
Consider that the end of str is not the end of a line and should not match operator ‘$’.

If rx was created with the regexp/newline option above, ‘$’ will still match before a newline in str.

;; Regexp to match uppercase letters
(define r (make-regexp "[A-Z]*"))

;; Regexp to match letters, ignoring case
(define ri (make-regexp "[A-Z]*" regexp/icase))

;; Search for bob using regexp r
(match:substring (regexp-exec r "bob"))
⇒ "" ; no match

;; Search for bob using regexp ri
(match:substring (regexp-exec ri "Bob"))
⇒ "Bob" ; matched case insensitive

regexp? obj  [Scheme Procedure]
scm_regexp_p (obj)  [C Function]
Return #t if obj is a compiled regular expression, or #f otherwise.

list-matches regexp str [flags]  [Scheme Procedure]
Return a list of match structures which are the non-overlapping matches of regexp in str. regexp can be either a pattern string or a compiled regexp. The flags argument is as per regexp-exec above.

(map match:substring (list-matches "[a-z]+" "abc 42 def 78"))
⇒ ("abc" "def")

fold-matches regexp str init proc [flags]  [Scheme Procedure]
Apply proc to the non-overlapping matches of regexp in str, to build a result. regexp can be either a pattern string or a compiled regexp. The flags argument is as per regexp-exec above.

proc is called as (proc match prev) where match is a match structure and prev is the previous return from proc. For the first call prev is the given init parameter. fold-matches returns the final value from proc.

For example to count matches,

(fold-matches "[a-z][0-9]" "abc x1 def y2" 0
 (lambda (match count)
   (1+ count)))
⇒ 2

Regular expressions are commonly used to find patterns in one string and replace them with the contents of another string. The following functions are convenient ways to do this.

regexp-substitute port match item . . .  [Scheme Procedure]
Write to port selected parts of the match structure match. Or if port is #f then form a string from those parts and return that.

Each item specifies a part to be written, and may be one of the following,

- A string. String arguments are written out verbatim.
- An integer. The submatch with that number is written (match:substring). Zero is the entire match.
The symbol 'pre'. The portion of the matched string preceding the regexp match is written (match:prefix).

The symbol 'post'. The portion of the matched string following the regexp match is written (match:suffix).

For example, changing a match and retaining the text before and after,

```scheme
(regexp-substitute #f (string-match "[0-9]+" "number 25 is good")
  'pre "37" 'post)
⇒ "number 37 is good"
```

Or matching a YYYYMMDD format date such as ‘20020828’ and re-ordering and hyphenating the fields.

```scheme
(define date-regex
 "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute #f (string-match date-regex s)
  'pre 2 "-" 3 "-" 1 'post " (" 0 ")")
⇒ "Date 04-29-2002 12am. (20020429)"
```

`regexp-substitute/global port regexp target item...` [Scheme Procedure]

Write to `port` selected parts of matches of `regexp` in `target`. If `port` is `#f` then form a string from those parts and return that. `regexp` can be a string or a compiled regex.

This is similar to `regexp-substitute`, but allows global substitutions on `target`. Each `item` behaves as per `regexp-substitute`, with the following differences,

- A function. Called as `(item match)` with the match structure for the `regexp` match, it should return a string to be written to `port`.
- The symbol 'post'. This doesn’t output anything, but instead causes `regexp-substitute/global` to recurse on the unmatched portion of `target`.

This `must` be supplied to perform a global search and replace on `target`; without it `regexp-substitute/global` returns after a single match and output.

For example, to collapse runs of tabs and spaces to a single hyphen each,

```scheme
(regexp-substitute/global #f "[ \t]+" "this is the text"
  'pre "-" 'post)
⇒ "this-is-the-text"
```

Or using a function to reverse the letters in each word,

```scheme
(regexp-substitute/global #f "[a-z]+" "to do and not-do"
  'pre (lambda (m) (string-reverse (match:substring m))) 'post)
⇒ "ot od dna ton-od"
```

Without the `post` symbol, just one regexp match is made. For example the following is the date example from `regexp-substitute` above, without the need for the separate `string-match` call.

```scheme
(define date-regex
 "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute/global #f date-regex s
  'pre 2 "-" 3 "-" 1 'post " (" 0 ")")
⇒ "Date 04-29-2002 12am. (20020429)"
```
6.15.2 Match Structures

A match structure is the object returned by string-match and regexp-exec. It describes which portion of a string, if any, matched the given regular expression. Match structures include: a reference to the string that was checked for matches; the starting and ending positions of the regexp match; and, if the regexp included any parenthesized subexpressions, the starting and ending positions of each submatch.

In each of the regexp match functions described below, the match argument must be a match structure returned by a previous call to string-match or regexp-exec. Most of these functions return some information about the original target string that was matched against a regular expression; we will call that string target for easy reference.

\[ \text{regexp-match? \ obj} \]

[Scheme Procedure]
Return \#t if obj is a match structure returned by a previous call to regexp-exec, or \#f otherwise.

\[ \text{match:substring \ match \ [n]} \]

[Scheme Procedure]
Return the portion of target matched by subexpression number \(n\). Submatch 0 (the default) represents the entire regexp match. If the regular expression as a whole matched, but the subexpression number \(n\) did not match, return \#f.

\[
\begin{align*}
\text{(define s (string-match \"[0-9][0-9][0-9][0-9]\" \"blah2002foo\")}) \\
\text{(match:substring s)} \\
\Rightarrow \text{"2002"}
\end{align*}
\]

;; match starting at offset 6 in the string
\[
\begin{align*}
\text{(match:substring \ (string-match \"[0-9][0-9][0-9][0-9]\" \"blah987654\" \ 6))} \\
\Rightarrow \text{"7654"}
\end{align*}
\]

\[ \text{match:start \ match \ [n]} \]

[Scheme Procedure]
Return the starting position of submatch number \(n\).

In the following example, the result is 4, since the match starts at character index 4:

\[
\begin{align*}
\text{(define s (string-match \"[0-9][0-9][0-9][0-9]\" \"blah2002foo\")}) \\
\text{(match:start s)} \\
\Rightarrow 4
\end{align*}
\]

\[ \text{match:end \ match \ [n]} \]

[Scheme Procedure]
Return the ending position of submatch number \(n\).

In the following example, the result is 8, since the match runs between characters 4 and 8 (i.e. the “2002”).

\[
\begin{align*}
\text{(define s (string-match \"[0-9][0-9][0-9][0-9]\" \"blah2002foo\")}) \\
\text{(match:end s)} \\
\Rightarrow 8
\end{align*}
\]
match:prefix match

Return the unmatched portion of target preceding the regexp match.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:prefix s)
⇒ "blah"
```

match:suffix match

Return the unmatched portion of target following the regexp match.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:suffix s)
⇒ "foo"
```

match:count match

Return the number of parenthesized subexpressions from match. Note that the entire regular expression match itself counts as a subexpression, and failed submatches are included in the count.

```
(match:count s)
```

match:string match

Return the original target string.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:string s)
⇒ "blah2002foo"
```

### 6.15.3 Backslash Escapes

Sometimes you will want a regexp to match characters like ‘*’ or ‘$’ exactly. For example, to check whether a particular string represents a menu entry from an Info node, it would be useful to match it against a regexp like ‘^*[^:\]*::’. However, this won’t work; because the asterisk is a metacharacter, it won’t match the ‘*’ at the beginning of the string. In this case, we want to make the first asterisk un-magic.

You can do this by preceding the metacharacter with a backslash character ‘\’. (This is also called quoting the metacharacter, and is known as a backslash escape.) When Guile sees a backslash in a regular expression, it considers the following glyph to be an ordinary character, no matter what special meaning it would ordinarily have. Therefore, we can make the above example work by changing the regexp to ‘\^*[^:\]*::’. The ‘\^’ sequence tells the regular expression engine to match only a single asterisk in the target string.

Since the backslash is itself a metacharacter, you may force a regexp to match a backslash in the target string by preceding the backslash with itself. For example, to find variable references in a TeX program, you might want to find occurrences of the string ‘\let\[A-Za-z]*’ would do this: the double backslashes in the regexp each match a single backslash in the target string.

regexp-quote str

Quote each special character found in str with a backslash, and return the resulting string.
Very important: Using backslash escapes in Guile source code (as in Emacs Lisp or C) can be tricky, because the backslash character has special meaning for the Guile reader. For example, if Guile encounters the character sequence ‘\n’ in the middle of a string while processing Scheme code, it replaces those characters with a newline character. Similarly, the character sequence ‘\t’ is replaced by a horizontal tab. Several of these escape sequences are processed by the Guile reader before your code is executed. Unrecognized escape sequences are ignored: if the characters ‘\*’ appear in a string, they will be translated to the single character ‘*’.

This translation is obviously undesirable for regular expressions, since we want to be able to include backslashes in a string in order to escape regexp metacharacters. Therefore, to make sure that a backslash is preserved in a string in your Guile program, you must use two consecutive backslashes:

```scheme
(define Info-menu-entry-pattern (make-regexp "^-\* [^:]*"))
```

The string in this example is preprocessed by the Guile reader before any code is executed. The resulting argument to `make-regexp` is the string ‘^-\* [^:]*’, which is what we really want.

This also means that in order to write a regular expression that matches a single backslash character, the regular expression string in the source code must include four backslashes. Each consecutive pair of backslashes gets translated by the Guile reader to a single backslash, and the resulting double-backslash is interpreted by the regexp engine as matching a single backslash character. Hence:

```scheme
(define tex-variable-pattern (make-regexp "\\let\\=[A-Za-z]*"))
```

The reason for the unwieldiness of this syntax is historical. Both regular expression pattern matchers and Unix string processing systems have traditionally used backslashes with the special meanings described above. The POSIX regular expression specification and ANSI C standard both require these semantics. Attempting to abandon either convention would cause other kinds of compatibility problems, possibly more severe ones. Therefore, without extending the Scheme reader to support strings with different quoting conventions (an ungainly and confusing extension when implemented in other languages), we must adhere to this cumbersome escape syntax.

### 6.16 LALR(1) Parsing

The (system base lalr) module provides the lalr-scm LALR(1) parser generator by Dominique Boucher ([https://github.com/schemeway/lalr-scm/](https://github.com/schemeway/lalr-scm/)). lalr-scm uses the same algorithm as GNU Bison (see Section “Introduction” in Bison, The Yacc-compatible Parser Generator). Parsers are defined using the lalr-parser macro.

**lalr-parser [options] tokens rules...**  
[Scheme Syntax]

Generate an LALR(1) syntax analyzer. `tokens` is a list of symbols representing the terminal symbols of the grammar. `rules` are the grammar production rules.

Each rule has the form `(non-terminal (rhs ...)) : action ...`, where `non-terminal` is the name of the rule, `rhs` are the right-hand sides, i.e., the production rule, and `action` is a semantic action associated with the rule.

The generated parser is a two-argument procedure that takes a `tokenizer` and a `syntax error procedure`. The tokenizer should be a thunk that returns lexical tokens as
produced by `make-lexical-token`. The syntax error procedure may be called with at least an error message (a string), and optionally the lexical token that caused the error.

Please refer to the `lalr-scm` documentation for details.

## 6.17 PEG Parsing

Parsing Expression Grammars (PEGs) are a way of specifying formal languages for text processing. They can be used either for matching (like regular expressions) or for building recursive descent parsers (like lex/yacc). Guile uses a superset of PEG syntax that allows more control over what information is preserved during parsing.

Wikipedia has a clear and concise introduction to PEGs if you want to familiarize yourself with the syntax: [http://en.wikipedia.org/wiki/Parsing_expression_grammar](http://en.wikipedia.org/wiki/Parsing_expression_grammar).

The `ice-9 peg` module works by compiling PEGs down to lambda expressions. These can either be stored in variables at compile-time by the define macros (`define-peg-pattern` and `define-peg-string-patterns`) or calculated explicitly at runtime with the compile functions (`compile-peg-pattern` and `peg-string-compile`).

They can then be used for either parsing (`match-pattern`) or searching (`search-for-pattern`). For convenience, `search-for-pattern` also takes pattern literals in case you want to inline a simple search (people often use regular expressions this way).

The rest of this documentation consists of a syntax reference, an API reference, and a tutorial.

### 6.17.1 PEG Syntax Reference

**Normal PEG Syntax:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence a b</code></td>
<td>Parses <code>a</code>. If this succeeds, continues to parse <code>b</code> from the end of the text parsed as <code>a</code>. Succeeds if both <code>a</code> and <code>b</code> succeed.</td>
</tr>
<tr>
<td>&quot;a b&quot;</td>
<td></td>
</tr>
<tr>
<td>(and a b)</td>
<td></td>
</tr>
<tr>
<td><code>ordered choice a b</code></td>
<td>Parses <code>a</code>. If this fails, backtracks and parses <code>b</code>. Succeeds if either <code>a</code> or <code>b</code> succeeds.</td>
</tr>
<tr>
<td>&quot;a/b&quot;</td>
<td></td>
</tr>
<tr>
<td>(or a b)</td>
<td></td>
</tr>
<tr>
<td><code>zero or more a</code></td>
<td>Parses <code>a</code> as many times in a row as it can, starting each <code>a</code> at the end of the text parsed by the previous <code>a</code>. Always succeeds.</td>
</tr>
<tr>
<td>&quot;a*&quot;</td>
<td></td>
</tr>
<tr>
<td>(* a)</td>
<td></td>
</tr>
</tbody>
</table>
one or more a
Parses a as many times in a row as it can, starting each a at the end of the text parsed by the previous a. Succeeds if at least one a was parsed.
"a+"
(+ a)

optional a
Tries to parse a. Succeeds if a succeeds.
"a?"
(? a)

followed by a
Makes sure it is possible to parse a, but does not actually parse it. Succeeds if a would succeed.
"&a"
(followed-by a)

not followed by a
Makes sure it is impossible to parse a, but does not actually parse it. Succeeds if a would fail.
"!a"
(not-followed-by a)

string literal “abc”
Parses the string "abc". Succeeds if that parsing succeeds.
"'abc'"
"abc"

any character
Parses any single character. Succeeds unless there is no more text to be parsed.
"."

character class a b
Alternative syntax for “Ordered Choice a b” if a and b are characters.
"[ab]"
(or "a" "b")

range of characters a z
Parses any character falling between a and z.
"[a-z]"
(range #\a #\z)
Example:

"(a !b / c &d*) 'e'+"

Would be:

(and
 (or
  (and a (not-followed-by b))
  (and c (followed-by (* d))))
(+ "e")

Extended Syntax

There is some extra syntax for S-expressions.

ignore a

Ignore the text matching a

[PEG Pattern]

capture a

Capture the text matching a.

[PEG Pattern]

peg a

Embed the PEG pattern a using string syntax.

Example:

"!a / 'b'"

Is equivalent to

(or (peg "$a") "b")

and

(or (not-followed-by a) "b")

6.17.2 PEG API Reference

Define Macros

The most straightforward way to define a PEG is by using one of the define macros (both of these macroexpand into define expressions). These macros bind parsing functions to variables. These parsing functions may be invoked by match-pattern or search-for-pattern, which return a PEG match record. Raw data can be retrieved from this record with the PEG match deconstructor functions. More complicated (and perhaps enlightening) examples can be found in the tutorial.

define-peg-string-patterns peg-string

[Scheme Macro]

Defines all the nonterminals in the PEG peg-string. More precisely, define-peg-string-patterns takes a superset of PEGs. A normal PEG has a <-> between the nonterminal and the pattern. define-peg-string-patterns uses this symbol to determine what information it should propagate up the parse tree. The normal <-> propagates the matched text up the parse tree, <- propagates the matched text up the parse tree tagged with the name of the nonterminal, and < discards that matched text and propagates nothing up the parse tree. Also, nonterminals may consist of any
alphanumeric character or a “-” character (in normal PEGs nonterminals can only be alphabetic).

For example, if we:

\[
\begin{align*}
(\text{define-peg-string-patterns} & \ \\
  \text{"as "} & \text{\text{\textquoteleft a\textquotesingle\textquotesingle}+} \\
  \text{bs} & \text{\text{\textquoteleft b\textquotesingle\textquotesingle}+} \\
  \text{as-or-bs} & \text{\text{\textquoteleft as\textquotesingle\textquotesingle}/\text{\textquoteleft bs\textquotesingle\textquotesingle}}) \\
(\text{define-peg-string-patterns} & \ \\
  \text{"as-tag \text{\textquoteleft a\textquotesingle\textquotesingle}+} \\
  \text{bs-tag} & \text{\text{\textquoteleft b\textquotesingle\textquotesingle}+} \\
  \text{as-or-bs-tag} & \text{\text{\textquoteleft as\textquotesingle\textquotesingle}/\text{\textquoteleft bs\textquotesingle\textquotesingle}})
\end{align*}
\]

Then:

\[
\begin{align*}
(\text{match-pattern} & \text{ as-or-bs \text{\textquoteleft aabbcc\textquoteright\textquoteright}}) \Rightarrow \\
\#<\text{peg start: 0 end: 2 string: aabbcc tree: aa}> \\
(\text{match-pattern} & \text{ as-or-bs-tag \text{\textquoteleft aabbcc\textquoteright\textquoteright}}) \Rightarrow \\
\#<\text{peg start: 0 end: 2 string: aabbcc tree: (as-or-bs-tag (as-tag aa))}>
\end{align*}
\]

Note that in doing this, we have bound 6 variables at the toplevel (as, bs, as-or-bs, as-tag, bs-tag, and as-or-bs-tag).

**define-peg-pattern**  
*name* capture-type *peg-sexp*  

[Scheme Macro]  
Defines a single nonterminal *name*. capture-type determines how much information is passed up the parse tree. *peg-sexp* is a PEG in S-expression form.

Possible values for capture-type:

**all** passes the matched text up the parse tree tagged with the name of the nonterminal.

**body** passes the matched text up the parse tree.

**none** passes nothing up the parse tree.

For Example, if we:

\[
\begin{align*}
(\text{define-peg-pattern} & \text{ as body (\text{\textquoteleft +}\text{\textquoteleft a\textquoteright\textquoteright})} \\
(\text{define-peg-pattern} & \text{ bs body (\text{\textquoteleft +}\text{\textquoteleft b\textquoteright\textquoteright})} \\
(\text{define-peg-pattern} & \text{ as-or-bs body (or as bs)}) \\
(\text{define-peg-pattern} & \text{ as-tag all (\text{\textquoteleft +}\text{\textquoteleft a\textquoteright\textquoteright})} \\
(\text{define-peg-pattern} & \text{ bs-tag all (\text{\textquoteleft +}\text{\textquoteleft b\textquoteright\textquoteright})} \\
(\text{define-peg-pattern} & \text{ as-or-bs-tag all (or as-tag bs-tag)})
\end{align*}
\]

Then:

\[
\begin{align*}
(\text{match-pattern} & \text{ as-or-bs \text{\textquoteleft aabbcc\textquoteright\textquoteright}}) \Rightarrow \\
\#<\text{peg start: 0 end: 2 string: aabbcc tree: aa}> \\
(\text{match-pattern} & \text{ as-or-bs-tag \text{\textquoteleft aabbcc\textquoteright\textquoteright}) \Rightarrow \\
\#<\text{peg start: 0 end: 2 string: aabbcc tree: (as-or-bs-tag (as-tag aa))}>
\end{align*}
\]

Note that in doing this, we have bound 6 variables at the toplevel (as, bs, as-or-bs, as-tag, bs-tag, and as-or-bs-tag).
Compile Functions

It is sometimes useful to be able to compile anonymous PEG patterns at runtime. These functions let you do that using either syntax.

**peg-string-compile**  
**peg-string capture-type**  
[Scheme Procedure]  
Compiles the PEG pattern in `peg-string` propagating according to `capture-type` (capture-type can be any of the values from `define-peg-pattern`).

**compile-peg-pattern**  
**peg-sexp capture-type**  
[Scheme Procedure]  
Compiles the PEG pattern in `peg-sexp` propagating according to `capture-type` (capture-type can be any of the values from `define-peg-pattern`).

The functions return syntax objects, which can be useful if you want to use them in macros. If all you want is to define a new nonterminal, you can do the following:

```
(define exp '(+ "a"))
(define as (compile (compile-peg-pattern exp 'body)))
```

You can use this nonterminal with all of the regular PEG functions:

```
(match-pattern as "aaaaa") ⇒
#<peg start: 0 end: 5 string: bbbbb tree: bbbbb>
```

Parsing & Matching Functions

For our purposes, “parsing” means parsing a string into a tree starting from the first character, while “matching” means searching through the string for a substring. In practice, the only difference between the two functions is that `match-pattern` gives up if it can’t find a valid substring starting at index 0 and `search-for-pattern` keeps looking. They are both equally capable of “parsing” and “matching” given those constraints.

**match-pattern**  
**nonterm string**  
[Scheme Procedure]  
Parses `string` using the PEG stored in `nonterm`. If no match was found, `match-pattern` returns false. If a match was found, a PEG match record is returned.

The `capture-type` argument to `define-peg-pattern` allows you to choose what information to hold on to while parsing. The options are:

- `all`  
  tag the matched text with the nonterminal

- `body`  
  just the matched text

- `none`  
  nothing

```
(define-peg-pattern as all (+ "a"))
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as aa)>
```

```
(define-peg-pattern as body (+ "a"))
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
```

```
(define-peg-pattern as none (+ "a"))
(match-pattern as "aabbcc") ⇒
```

search-for-pattern nonterm-or-peg string

[Scheme Macro]
Searches through string looking for a matching subexpression. nonterm-or-peg can
either be a nonterminal or a literal PEG pattern. When a literal PEG pattern is pro-
vided, search-for-pattern works very similarly to the regular expression searches
many hackers are used to. If no match was found, search-for-pattern returns false.
If a match was found, a PEG match record is returned.

PEG Match Records
The match-pattern and search-for-pattern functions both return PEG match records.
Actual information can be extracted from these with the following functions.

peg:string match-record
Returns the original string that was parsed in the creation of match-record.
**peg:start** *match-record*  
Returns the index of the first parsed character in the original string (from *peg:string*). If this is the same as *peg:end*, nothing was parsed.

**peg:end** *match-record*  
Returns one more than the index of the last parsed character in the original string (from *peg:string*). If this is the same as *peg:start*, nothing was parsed.

**peg:substring** *match-record*  
Returns the substring parsed by *match-record*. This is equivalent to 

\[(\text{substring} \ (\text{peg:string} \ \text{match-record}) \ \text{peg:start} \ \text{match-record}) \ \text{peg:end} \ \text{match-record})\].

**peg:tree** *match-record*  
Returns the tree parsed by *match-record*.

**peg-record?** *match-record*  
Returns true if *match-record* is a PEG match record, or false otherwise.

Example:

```scheme
(define-peg-pattern bs all (peg "'b'+")
(search-for-pattern bs "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: (bs bb)>

(let ((pm (search-for-pattern bs "aabbcc")))
  '((string , (peg:string pm))
   (start , (peg:start pm))
   (end , (peg:end pm))
   (substring , (peg:substring pm))
   (tree , (peg:tree pm))
   (record? , (peg-record? pm))) ⇒
  ((string "aabbcc")
   (start 2)
   (end 4)
   (substring "bb")
   (tree (bs "bb"))
   (record? #t))
```

**Miscellaneous**

**context-flatten** *tst* *lst*  
Takes a predicate *tst* and a list *lst*. Flattens *lst* until all elements are either atoms or satisfy *tst*. If *lst* itself satisfies *tst*, (*list lst*) is returned (this is a flat list whose only element satisfies *tst*).

```scheme
(context-flatten (lambda (x) (and (number? (car x)) (= (car x) 1))) '(2 2 (1 1 (2 2)) 2 2 (1 1))
  (context-flatten (lambda (x) (and (number? (car x)) (= (car x) 1))) '(1 1 (1 2))
```
keyword-flatten terms lst

A less general form of context-flatten. Takes a list of terminal atoms terms and flattens lst until all elements are either atoms, or lists which have an atom from terms as their first element.

(keyword-flatten '(a b) '(c a b (a c) (b c) (c (b a) (c a)))) ⇒
(c a b (a c) (b c) c (b a) c a)

If you’re wondering why this is here, take a look at the tutorial.

6.17.3 PEG Tutorial

Parsing /etc/passwd

This example will show how to parse /etc/passwd using PEGs.

First we define an example /etc/passwd file:

(define *etc-passwd*
"root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
messagebus:x:103:107::/var/run/dbus:/bin/false"
)

As a first pass at this, we might want to have all the entries in /etc/passwd in a list.

Doing this with string-based PEG syntax would look like this:

(define-peg-string-patterns
"passwd <- entry* !.
entry <-- (! NL .)* NL*
NL < '\n'")

A passwd file is 0 or more entries (entry*) until the end of the file (!. (. is any character, so !. means “not anything”)). We want to capture the data in the nonterminal passwd, but not tag it with the name, so we use <-.

An entry is a series of 0 or more characters that aren’t newlines ((! NL .)*) followed by 0 or more newlines (NL*). We want to tag all the entries with entry, so we use --.

A newline is just a literal newline ('\n'). We don’t want a bunch of newlines cluttering up the output, so we use < to throw away the captured data.

Here is the same PEG defined using S-expressions:

(define-peg-pattern passwd body (and (* entry) (not-followed-by peg-any)))
(define-peg-pattern entry all (and (* (and (not-followed-by NL) peg-any)) (* NL)))
(define-peg-pattern NL none "\n")

Obviously this is much more verbose. On the other hand, it’s more explicit, and thus easier to build automatically. However, there are some tricks that make S-expressions easier
to use in some cases. One is the `ignore` keyword; the string syntax has no way to say “throw away this text” except breaking it out into a separate nonterminal. For instance, to throw away the newlines we had to define `NL`. In the S-expression syntax, we could have simply written `(ignore "\n")`. Also, for the cases where string syntax is really much cleaner, the `peg` keyword can be used to embed string syntax in S-expression syntax. For instance, we could have written:

```lisp
(define-peg-pattern passwd body (peg "entry* !."))
```

However we define it, parsing `*etc-passwd*` with the `passwd` nonterminal yields the same results:

```lisp
(peg:tree (match-pattern passwd *etc-passwd*)) ⇒
((entry "root:x:0:0:root:/root:/bin/bash")
 (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
 (entry "bin:x:2:2:bin:/bin:/bin/sh")
 (entry "sys:x:3:3:sys:/dev:/bin/sh")
 (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
 (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
```

However, here is something to be wary of:

```lisp
(peg:tree (match-pattern passwd "one entry")) ⇒
(entry "one entry")
```

By default, the parse trees generated by PEGs are compressed as much as possible without losing information. It may not look like this is what you want at first, but uncompressed parse trees are an enormous headache (there’s no easy way to predict how deep particular lists will nest, there are empty lists littered everywhere, etc. etc.). One side-effect of this, however, is that sometimes the compressor is too aggressive. No information is discarded when `((entry "one entry"))` is compressed to `(entry "one entry")`, but in this particular case it probably isn’t what we want.

There are two functions for easily dealing with this: `keyword-flatten` and `context-flatten`. The `keyword-flatten` function takes a list of keywords and a list to flatten, then tries to coerce the list such that the first element of all sublists is one of the keywords. The `context-flatten` function is similar, but instead of a list of keywords it takes a predicate that should indicate whether a given sublist is good enough (refer to the API reference for more details).

What we want here is `keyword-flatten`.

```lisp
(keyword-flatten '(entry) (peg:tree (match-pattern passwd *etc-passwd*))) ⇒
((entry "root:x:0:0:root:/root:/bin/bash")
 (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
 (entry "bin:x:2:2:bin:/bin:/bin/sh")
 (entry "sys:x:3:3:sys:/dev:/bin/sh")
 (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
 (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
```

Of course, this is a somewhat contrived example. In practice we would probably just tag the `passwd` nonterminal to remove the ambiguity (using either the `all` keyword for S-expressions or the `<--` symbol for strings)..
(define-peg-pattern tag-passwd all (peg "entry* !."))
(peg:tree (match-pattern tag-passwd etc-passwd*)) ⇒
(tag-passwd
  (entry "root:x:0:0:root:/root:/bin/bash")
  (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
  (entry "bin:x:2:2:daemon:/usr/sbin:/bin/sh")
  (entry "sys:x:3:3:sys:/dev:/bin/sh")
  (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
  (entry "messagebus:x:103:107:/:var/run/dbus:/bin/false")
)
(peg:tree (match-pattern tag-passwd "one entry"))
(tag-passwd
  (entry "one entry")
)

If you're ever uncertain about the potential results of parsing something, remember the two absolute rules:

1. No parsing information will ever be discarded.
2. There will never be any lists with fewer than 2 elements.

For the purposes of (1), "parsing information" means things tagged with the any keyword or the <-- symbol. Plain strings will be concatenated.

Let's extend this example a bit more and actually pull some useful information out of the passwd file:

(define-peg-string-patterns
  "passwd <-- entry* !.
  entry <-- login C pass C uid C gid C nameORcomment C homedir C shell NL*
  login <-- text
  pass <-- text
  uid <-- [0-9]*
  gid <-- [0-9]*
  nameORcomment <-- text
  homedir <-- path
  shell <-- path
  path <-- (SLASH pathELEMENT)*
  pathELEMENT <-- (!NL !C !'/' .)*
  text <-- (!NL !C .)*
  C < ':'
  NL < '\n'
  SLASH < '/'")

This produces rather pretty parse trees:

(passwd
  (entry (login "root")
    (pass "x")
    (uid "0")
    (gid "0")
    (nameORcomment "root")
    (homedir (path (pathELEMENT "root")))
    (shell (path (pathELEMENT "bin") (pathELEMENT "bash")))))
(entry (login "daemon")
  (pass "x")
  (uid "1")
  (gid "1")
  (nameORcomment "daemon")
  (homedir
    (path (pathELEMENT "usr") (pathELEMENT "sbin")))
  (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "bin")
  (pass "x")
  (uid "2")
  (gid "2")
  (nameORcomment "bin")
  (homedir (path (pathELEMENT "bin")))
  (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "sys")
  (pass "x")
  (uid "3")
  (gid "3")
  (nameORcomment "sys")
  (homedir (path (pathELEMENT "dev")))
  (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "nobody")
  (pass "x")
  (uid "65534")
  (gid "65534")
  (nameORcomment "nobody")
  (homedir (path (pathELEMENT "nonexistent")))
  (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "messagebus")
  (pass "x")
  (uid "103")
  (gid "107")
  nameORcomment
  (homedir
    (path (pathELEMENT "var")
      (pathELEMENT "run")
      (pathELEMENT "dbus"))
    (shell (path (pathELEMENT "bin") (pathELEMENT "false")))))

Notice that when there's no entry in a field (e.g. nameORcomment for messagebus) the symbol is inserted. This is the “don't throw away any information” rule—we successfully matched a nameORcomment of 0 characters (since we used * when defining it). This is usually what you want, because it allows you to e.g. use list-ref to pull out elements (since they all have known offsets).

If you'd prefer not to have symbols for empty matches, you can replace the * with a + and add a ? after the nameORcomment in entry. Then it will try to parse 1 or more
characters, fail (inserting nothing into the parse tree), but continue because it didn’t have to match the nameORcomment to continue.

**Embedding Arithmetic Expressions**

We can parse simple mathematical expressions with the following PEG:

```
(define-peg-string-patterns
  "expr <- sum
  sum <-- (product ('+' / '-') sum) / product
  product <-- (value ('*' / '/') product) / value
  value <-- number / '(' expr ')' ,
  number <-- [0-9]+")
Then:

(peg:tree (match-pattern expr "1+1/2*3+(1+1)/2") ) ⇒
(sum (product (value (number "1")))
  "+"
  (sum (product
     (value (number "1"))
     "/"
     (product
      (value (number "2"))
      "*"
      (product (value (number "3"))))))
  "+"
  (sum (product
     (value "("
      (sum (product (value (number "1")))
      "+"
      (sum (product (value (number "1")))))
      ")")
      "/"
      (product (value (number "2")))
    ))

There is very little wasted effort in this PEG. The **number** nonterminal has to be tagged because otherwise the numbers might run together with the arithmetic expressions during the string concatenation stage of parse-tree compression (the parser will see “1” followed by “/” and decide to call it “1/”). When in doubt, tag.

It is very easy to turn these parse trees into lisp expressions:

```
  (define (parse-sum sum left . rest)
    (if (null? rest)
      (apply parse-product left)
      (list (string->symbol (car rest))
      (apply parse-product left)
      (apply parse-sum (cadr rest))))

  (define (parse-product product left . rest)
    (if (null? rest)
      (apply parse-sum product left . rest)
      (if (null? rest)

```
(apply parse-value left)
  (list (string->symbol (car rest))
  (apply parse-value left)
  (apply parse-product (cadr rest))))

(define (parse-value value first . rest)
  (if (null? rest)
      (string->number (cadr first))
      (apply parse-sum (car rest))))

(define parse-expr parse-sum)

(Notice all these functions look very similar; for a more complicated PEG, it would be
worth abstracting.)

Then:

(apply parse-expr (peg:tree (match-pattern expr "1+1/2*3+(1+1)/2"))) ⇒
(+ 1 (+ (/ 1 (* 2 3)) (/ (+ 1 1) 2)))

But wait! The associativity is wrong! Where it says (/ 1 (* 2 3)), it should say (* (/ 1 2) 3).

It’s tempting to try replacing e.g. "sum <-- (product ('+' / '-') sum) / product"
with "sum <-- (sum ('+' / '-' ) product) / product", but this is a Bad Idea. PEGs don’t
support left recursion. To see why, imagine what the parser will do here. When it tries to
parse sum, it first has to try and parse sum. But to do that, it first has to try and parse
sum. This will continue until the stack gets blown off.

So how does one parse left-associative binary operators with PEGs? Honestly, this is
one of their major shortcomings. There’s no general-purpose way of doing this, but here
the repetition operators are a good choice:

(use-modules (srfi srfi-1))

(define-peg-string-patterns
  "expr <- sum
  sum <- (product ('+' / '-')*) product
  product <- (value ('*' / '/')*) value
  value <- number / '(expr')
  number <- [0-9]+")

;; take a deep breath...

(define (make-left-parser next-func)
  (lambda (sum first . rest) ;; general form, comments below assume
    ;; that we’re dealing with a sum expression
    (if (null? rest) ;; form (sum (product ...))
      (apply next-func first)
      (if (string? (cadr first));; form (sum ((product ...) "+") (product ...))
        (list (string->symbol (cadr first))
        (apply next-func (car first))
        (apply next-func (car rest)))
      (list (string->symbol (cadr first))
      (apply next-func (car first))
      (apply next-func (car rest)))))
;;; form (sum (((product ...) "+") ((product ...) "+")) (product ...))
(car
(reduce
(lambda (l r)
  (list (list (cadr r) (car r) (apply next-func (car l)))
    (string->symbol (cadr l))))
  'ignore
(append
  (list (apply next-func (caar first))
    (string->symbol (cadar first))))
(cdr first)
  ;; the last one has to be added in
  (list (append rest '("done"))))))))

(define (parse-value value first . rest)
  (if (null? rest)
    (string->number (cadr first))
    (apply parse-sum (car rest))))
(define parse-product (make-left-parser parse-value))
(define parse-sum (make-left-parser parse-product))
(define parse-expr parse-sum)

Then:
(apply parse-expr (peg:tree (match-pattern expr "1+1/2*3+(1+1)/2"))) ⇒
(+ (+ 1 (* (/ 1 2) 3)) (/ (+ 1 1) 2))

As you can see, this is much uglier (it could be made prettier by using context-flatten, but the way it’s written above makes it clear how we deal with the three ways the zero-or-more * expression can parse). Fortunately, most of the time we can get away with only using right-associativity.

Simplified Functions
For a more tantalizing example, consider the following grammar that parses (highly) simplified C functions:

```
(define-peg-string-patterns
  "cfunc <-- cSP ctype cSP cname cSP cargs cLB cSP cbody cRB
cfunc <- cidentifer
cname <- cidentifer
cargs <- cLP (! (cSP cRP) carg cSP (cCOMMA / cRP) cSP)* cSP
carg <- cSP cfunc cSP cSP cname
cbody <- cstatement *
cidentifier <- [a-zA-Z][a-zA-Z0-9_]*
cstatement <- (!';'.)*cSC cSP
cSC < ';'
cCOMMA < ','
cLP < '('
cRP < ')'
```

cLB < '{'
cRB < '}'
cSP < [ \t\n\]*")

Then:

(match-pattern cfunc "int square(int a) { return a*a; }") ⇒
(32
  (cfunc (ctype "int")
    (cname "square")
    (cargs (carg (ctype "int") (cname "a")))
    (cbody (cstatement "return a*a"))))

And:

(match-pattern cfunc "int mod(int a, int b) { int c = a/b; return a-b*c; }") ⇒
(52
  (cfunc (ctype "int")
    (cname "mod")
    (cargs (carg (ctype "int") (cname "a"))
      (carg (ctype "int") (cname "b")))
    (cbody (cstatement "int c = a/b"
      (cstatement "return a- b*c")))))

By wrapping all the carg nonterminals in a cargs nonterminal, we were able to remove any ambiguity in the parsing structure and avoid having to call context-flatten on the output of match-pattern. We used the same trick with the cstatement nonterminals, wrapping them in a cbody nonterminal.

The whitespace nonterminal cSP used here is a (very) useful instantiation of a common pattern for matching syntactically irrelevant information. Since it’s tagged with < and ends with * it won’t clutter up the parse trees (all the empty lists will be discarded during the compression step) and it will never cause parsing to fail.

6.17.4 PEG Internals

A PEG parser takes a string as input and attempts to parse it as a given nonterminal. The key idea of the PEG implementation is that every nonterminal is just a function that takes a string as an argument and attempts to parse that string as its nonterminal. The functions always start from the beginning, but a parse is considered successful if there is material left over at the end.

This makes it easy to model different PEG parsing operations. For instance, consider the PEG grammar "ab", which could also be written (and "a" "b"). It matches the string “ab”. Here’s how that might be implemented in the PEG style:

(define (match-and-a-b str)
  (match-a str)
  (match-b str))

As you can see, the use of functions provides an easy way to model sequencing. In a similar way, one could model (or a b) with something like the following:

(define (match-or-a-b str)
  (or (match-a str) (match-b str)))
Here the semantics of a PEG or expression map naturally onto Scheme’s or operator. This function will attempt to run (match-a str), and return its result if it succeeds. Otherwise it will run (match-b str).

Of course, the code above wouldn’t quite work. We need some way for the parsing functions to communicate. The actual interface used is below.

**Parsing Function Interface**

A parsing function takes three arguments - a string, the length of that string, and the position in that string it should start parsing at. In effect, the parsing functions pass around substrings in pieces - the first argument is a buffer of characters, and the second two give a range within that buffer that the parsing function should look at.

Parsing functions return either #f, if they failed to match their nonterminal, or a list whose first element must be an integer representing the final position in the string they matched and whose cdr can be any other data the function wishes to return, or ’() if it doesn’t have any more data.

The one caveat is that if the extra data it returns is a list, any adjacent strings in that list will be appended by match-pattern. For instance, if a parsing function returns (13 ("a" "b" "c")), match-pattern will take (13 ("abc")) as its value.

For example, here is a function to match “ab” using the actual interface.

```scheme
(define (match-a-b str len pos)
  (and (< (+ pos 2) len)
       (string= str "ab" pos (+ pos 2))
       (list (+ pos 2) ’())) ; we return no extra information)
```

The above function can be used to match a string by running (match-pattern match-a-b "ab").

**Code Generators and Extensible Syntax**

PEG expressions, such as those in a define-peg-pattern form, are interpreted internally in two steps.

First, any string PEG is expanded into an s-expression PEG by the code in the (ice-9 peg string-peg) module.

Then, the s-expression PEG that results is compiled into a parsing function by the (ice-9 peg codegen) module. In particular, the function compile-peg-pattern is called on the s-expression. It then decides what to do based on the form it is passed.

The PEG syntax can be expanded by providing compile-peg-pattern more options for what to do with its forms. The extended syntax will be associated with a symbol, for instance my-parsing-form, and will be called on all PEG expressions of the form

```scheme
(my-parsing-form ...)
```

The parsing function should take two arguments. The first will be a syntax object containing a list with all of the arguments to the form (but not the form’s name), and the second will be the capture-type argument that is passed to define-peg-pattern.

New functions can be registered by calling (add-peg-compiler! symbol function), where symbol is the symbol that will indicate a form of this type and function is the code generating function described above. The function add-peg-compiler! is exported from the (ice-9 peg codegen) module.
6.18 Reading and Evaluating Scheme Code

This chapter describes Guile functions that are concerned with reading, loading, evaluating, and compiling Scheme code at run time.

6.18.1 Scheme Syntax: Standard and Guile Extensions

6.18.1.1 Expression Syntax

An expression to be evaluated takes one of the following forms.

- **symbol**
  A symbol is evaluated by dereferencing. A binding of that symbol is sought and the value there used. For example,
  
  ```scheme
  (define x 123)
  x ⇒ 123
  ```

- **(proc args...)**
  A parenthesised expression is a function call. `proc` and each argument are evaluated, then the function (which `proc` evaluated to) is called with those arguments.
  
  The order in which `proc` and the arguments are evaluated is unspecified, so be careful when using expressions with side effects.
  
  ```scheme
  (max 1 2 3) ⇒ 3
  ```

  ```scheme
  (define (get-some-proc) min)
  ((get-some-proc) 1 2 3) ⇒ 1
  ```

  The same sort of parenthesised form is used for a macro invocation, but in that case the arguments are not evaluated. See the descriptions of macros for more on this (see Section 6.10 [Macros], page 261, and see Section 6.10.2 [Syntax Rules], page 262).

- **constant**
  Number, string, character and boolean constants evaluate “to themselves”, so can appear as literals.
  
  ```scheme
  123 ⇒ 123
  99.9 ⇒ 99.9
  "hello" ⇒ "hello"
  #$z ⇒ #$z
  #t ⇒ #t
  ```

  Note that an application must not attempt to modify literal strings, since they may be in read-only memory.

- **(quote data)**
  Quoting is used to obtain a literal symbol (instead of a variable reference), a literal list (instead of a function call), or a literal vector. `'` is simply a shorthand for a quote form. For example,
  
  ```scheme
  'x ⇒ x
  '(1 2 3) ⇒ (1 2 3)
  '#(1 (2 3) 4) ⇒ #(1 (2 3) 4)
  (quote x) ⇒ x
  ```
Note that an application must not attempt to modify literal lists or vectors obtained from a `quote` form, since they may be in read-only memory.

`(quasiquote data)`

Backquote quasi-quotiation is like `quote`, but selected sub-expressions are evaluated. This is a convenient way to construct a list or vector structure most of which is constant, but at certain points should have expressions substituted. The same effect can always be had with suitable `list`, `cons` or `vector` calls, but quasi-quoting is often easier.

`(unquote expr)`

,`expr` Within the quasiquote data, `unquote` or , indicates an expression to be evaluated and inserted. The comma syntax , is simply a shorthand for an `unquote` form. For example,

```
'(1 2 (* 9 9) 3 4)  ⇒  (1 2 (* 9 9) 3 4)
'(1 2 ,(* 9 9) 3 4)  ⇒  (1 2 81 3 4)
'(1 (unquote (+ 1 1)) 3)  ⇒  (1 2 3)
'#(1 ,(/ 12 2))  ⇒  #(1 6)
```

`(unquote-splicing expr)`

,@`expr` Within the quasiquote data, `unquote-splicing` or ,@ indicates an expression to be evaluated and the elements of the returned list inserted. `expr` must evaluate to a list. The “comma-at” syntax ,@ is simply a shorthand for an `unquote-splicing` form.

```
(define x '(2 3))
'(1 ,x 4)  ⇒  (1 (2 3) 4)
'(1 ,@x 4)  ⇒  (1 2 3 4)
'(1 (unquote-splicing (map 1+ x)))  ⇒  (1 3 4)
'#(9 ,@x 9)  ⇒  #(9 2 3 9)
```

Notice ,@ differs from plain , in the way one level of nesting is stripped. For ,@ the elements of a returned list are inserted, whereas with , it would be the list itself inserted.

### 6.18.1.2 Comments

Comments in Scheme source files are written by starting them with a semicolon character (;). The comment then reaches up to the end of the line. Comments can begin at any column, and the may be inserted on the same line as Scheme code.

```
; Comment
;; Comment too
(define x 1)     ; Comment after expression
(let ((y 1))
  ;; Display something.
  (display y)
;;; Comment at left margin.
  (display (+ y 1))
```
It is common to use a single semicolon for comments following expressions on a line, to use two semicolons for comments which are indented like code, and three semicolons for comments which start at column 0, even if they are inside an indented code block. This convention is used when indenting code in Emacs’ Scheme mode.

### 6.18.1.3 Block Comments

In addition to the standard line comments defined by R5RS, Guile has another comment type for multiline comments, called block comments. This type of comment begins with the character sequence `#!` and ends with the characters `!#`.

These comments are compatible with the block comments in the Scheme Shell `scsh` (see Section 7.17 [The Scheme shell (scsh)], page 728). The characters `#!` were chosen because they are the magic characters used in shell scripts for indicating that the name of the program for executing the script follows on the same line.

Thus a Guile script often starts like this.

```
#! /usr/local/bin/guile -s
!#
```

More details on Guile scripting can be found in the scripting section (see Section 4.3 [Guile Scripting], page 41).

Similarly, Guile (starting from version 2.0) supports nested block comments as specified by R6RS and SRFI-30 (http://srfi.schemers.org/srfi-30/srfi-30.html):

```
(+ 1 #| this is a #| nested |# block comment |# 2)
⇒ 3
```

For backward compatibility, this syntax can be overridden with `read-hash-extend` (see Section 6.18.1.6 [Reader Extensions], page 382).

There is one special case where the contents of a comment can actually affect the interpretation of code. When a character encoding declaration, such as `coding: utf-8` appears in one of the first few lines of a source file, it indicates to Guile’s default reader that this source code file is not ASCII. For details see Section 6.18.8 [Character Encoding of Source Files], page 392.

### 6.18.1.4 Case Sensitivity

Scheme as defined in R5RS is not case sensitive when reading symbols. Guile, on the contrary is case sensitive by default, so the identifiers `guile-whuzzy` and `Guile-Whuzzy` are the same in R5RS Scheme, but are different in Guile.

It is possible to turn off case sensitivity in Guile by setting the reader option `case-insensitive`. For more information on reader options, see Section 6.18.2 [Scheme Read], page 382.

```
(read-enable 'case-insensitive)
```

It is also possible to disable (or enable) case sensitivity within a single file by placing the reader directives `#!fold-case` (or `#!no-fold-case`) within the file itself.

### 6.18.1.5 Keyword Syntax
6.18.1.6 Reader Extensions

`read-hash-extend chr proc`  
[Scheme Procedure]

`scm_read_hash_extend (chr, proc)`  
[C Function]

Install the procedure `proc` for reading expressions starting with the character sequence `#` and `chr`. `proc` will be called with two arguments: the character `chr` and the port to read further data from. The object returned will be the return value of `read`. Passing `#f` for `proc` will remove a previous setting.

6.18.2 Reading Scheme Code

`read [port]`  
[Scheme Procedure]

`scm_read (port)`  
[C Function]

Read an s-expression from the input port `port`, or from the current input port if `port` is not specified. Any whitespace before the next token is discarded.

The behaviour of Guile’s Scheme reader can be modified by manipulating its read options.

`read-options [setting]`  
[Scheme Procedure]

Display the current settings of the global read options. If `setting` is omitted, only a short form of the current read options is printed. Otherwise if `setting` is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `read-options` at the prompt.

```
scheme@(guile-user)> (read-options)
(square-brackets keywords #f positions)
scheme@(guile-user)> (read-options 'help)
copy no Copy source code expressions.
positions yes Record positions of source code expressions.
case-insensitive no Convert symbols to lower case.
keywords #f Style of keyword recognition: #f, 'prefix or 'postfix.
r6rs-hex-escapes no Use R6RS variable-length character and string hex escapes.
square-brackets yes Treat '[' and ']' as parentheses, for R6RS compatibility.
hungry-eol-escapes no In strings, consume leading whitespace after an escaped end-of-line.
curly-infix no Support SRFI-105 curly infix expressions.
r7rs-symbols no Support R7RS |...| symbol notation.
```

Note that Guile also includes a preliminary mechanism for setting read options on a per-port basis. For instance, the `case-insensitive` read option is set (or unset) on the port when the reader encounters the `#!fold-case` or `#!no-fold-case` reader directives. Similarly, the `#!curly-infix` reader directive sets the curly-infix read option on the port, and `#!curly-infix-and-bracket-lists` sets curly-infix and unsets square-brackets on the port (see Section 7.5.44 [SRFI-105], page 647). There is currently no other way to access or set the per-port read options.

The boolean options may be toggled with `read-enable` and `read-disable`. The non-boolean `keywords` option must be set using `read-set!`.

`read-enable option-name`  
[Scheme Procedure]

`read-disable option-name`  
[Scheme Procedure]
read-set! option-name value  

Modify the read options. read-enable should be used with boolean options and switches them on, read-disable switches them off.

read-set! can be used to set an option to a specific value. Due to historical oddities, it is a macro that expects an unquoted option name.

For example, to make read fold all symbols to their lower case (perhaps for compatibility with older Scheme code), you can enter:

(read-enable 'case-insensitive)

For more information on the effect of the r6rs-hex-escapes and hungry-eol-escapes options, see (see Section 6.6.5.1 [String Syntax], page 141).

For more information on the r7rs-symbols option, see (see Section 6.6.6.6 [Symbol Read Syntax], page 171).

6.18.3 Writing Scheme Values

Any scheme value may be written to a port. Not all values may be read back in (see Section 6.18.2 [Scheme Read], page 382), however.

write obj [port]  

Send a representation of obj to port or to the current output port if not given.

The output is designed to be machine readable, and can be read back with read (see Section 6.18.2 [Scheme Read], page 382). Strings are printed in double quotes, with escapes if necessary, and characters are printed in ‘#\’ notation.

display obj [port]  

Send a representation of obj to port or to the current output port if not given.

The output is designed for human readability, it differs from write in that strings are printed without double quotes and escapes, and characters are printed as per write-char, not in ‘#\’ form.

As was the case with the Scheme reader, there are a few options that affect the behavior of the Scheme printer.

print-options [setting]  

Display the current settings of the read options. If setting is omitted, only a short form of the current read options is printed. Otherwise if setting is the symbol help, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking print-options at the prompt.

scheme@(guile-user)> (print-options)

(quote-keywordish-symbols reader highlight-suffix "}" highlight-prefix "{"

scheme@(guile-user)> (print-options 'help)

highlight-prefix { The string to print before highlighted values.
highlight-suffix } The string to print after highlighted values.
quote-keywordish-symbols reader How to print symbols that have a colon as their first or last character. The value '#f' does not quote the colons; '#t' quotes them; 'reader' quotes them
when the reader option 'keywords' is not 'f'.

escape-newlines yes Render newlines as \n when printing using 'write'.

r7rs-symbols no Escape symbols using R7RS |...| symbol notation.

These options may be modified with the print-set! syntax.

**print-set! option-name value**  
[Scheme Syntax]  
Modify the print options. Due to historical oddities, print-set! is a macro that expects an unquoted option name.

### 6.18.4 Procedures for On the Fly Evaluation

Scheme has the lovely property that its expressions may be represented as data. The eval procedure takes a Scheme datum and evaluates it as code.

**eval exp module_or_state**  
[Scheme Procedure]
**scm_eval (exp, module_or_state)**  
[C Function]

Evaluate exp, a list representing a Scheme expression, in the top-level environment specified by module_or_state. While exp is evaluated (using primitive-eval), module_or_state is made the current module. The current module is reset to its previous value when eval returns. XXX - dynamic states. Example: (eval '(+ 1 2) (interaction-environment))

**interaction-environment**  
[Scheme Procedure]
**scm_interaction_environment ()**  
[C Function]

Return a specifier for the environment that contains implementation-defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

See Section 6.20.11 [Environments], page 421, for other environments.

One does not always receive code as Scheme data, of course, and this is especially the case for Guile's other language implementations (see Section 6.24 [Other Languages], page 455). For the case in which all you have is a string, we have eval-string. There is a legacy version of this procedure in the default environment, but you really want the one from (ice-9 eval-string), so load it up:

```scheme
(use-modules (ice-9 eval-string))
```

**eval-string string [#:module=#f] [#:file=#f] [#:line=#f] [#:column=#f] [#:lang=(current-language)] [#:compile?=#f]**  
[Scheme Procedure]

Parse string according to the current language, normally Scheme. Evaluate or compile the expressions it contains, in order, returning the last expression.

If the module keyword argument is set, save a module excursion (see Section 6.20.8 [Module System Reflection], page 416) and set the current module to module before evaluation.

The file, line, and column keyword arguments can be used to indicate that the source string begins at a particular source location.
Finally, `lang` is a language, defaulting to the current language, and the expression is compiled if `compile?` is true or there is no evaluator for the given language.

```c
scm_eval_string (string) [C Function]
scm_eval_string_in_module (string, module) [C Function]
These C bindings call `eval-string` from `(ice-9 eval-string)`, evaluating within `module` or the current module.
```

```scheme
apply proc arg ... arglst [Scheme Procedure]
scm_apply_0 (proc, arglst) [C Function]
scm_apply_1 (proc, arg1, arglst) [C Function]
scm_apply_2 (proc, arg1, arg2, arglst) [C Function]
scm_apply_3 (proc, arg1, arg2, arg3, arglst) [C Function]
scm_apply (proc, arg, rest) [C Function]
Call proc with arguments arg ... and the elements of the arglst list.

scm_apply takes parameters corresponding to a Scheme level `(lambda (proc arg1 . rest) ...)`. So arg1 and all but the last element of the rest list make up arg ..., and the last element of rest is the arglst list. Or if rest is the empty list `SCM_EOL` then there’s no arg ..., and (arg1) is the arglst.

arglst is not modified, but the rest list passed to `scm_apply` is modified.
```

```c
scm_call_0 (proc) [C Function]
scm_call_1 (proc, arg1) [C Function]
scm_call_2 (proc, arg1, arg2) [C Function]
scm_call_3 (proc, arg1, arg2, arg3) [C Function]
scm_call_4 (proc, arg1, arg2, arg3, arg4) [C Function]
scm_call_5 (proc, arg1, arg2, arg3, arg4, arg5) [C Function]
scm_call_6 (proc, arg1, arg2, arg3, arg4, arg5, arg6) [C Function]
scm_call_7 (proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7) [C Function]
scm_call_8 (proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8) [C Function]
scm_call_9 (proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9) [C Function]
Call proc with the given arguments.
```

```c
scm_call (proc, ...) [C Function]
Call proc with any number of arguments. The argument list must be terminated by `SCM_UNDEFINED`. For example:

```c
    scm_call (scm_c_public_ref ("guile", "+"),
              scm_from_int (1),
              scm_from_int (2),
              SCM_UNDEFINED);
```

```c
scm_call_n (proc, argv, nargs) [C Function]
Call proc with the array of arguments argv, as a SCM*. The length of the arguments should be passed in nargs, as a size_t.
primitive-eval exp

scm_primitive_eval (exp)

Evaluate exp in the top-level environment specified by the current module.

6.18.5 Compiling Scheme Code

The eval procedure directly interprets the S-expression representation of Scheme. An alternate strategy for evaluation is to determine ahead of time what computations will be necessary to evaluate the expression, and then use that recipe to produce the desired results. This is known as compilation.

While it is possible to compile simple Scheme expressions such as (+ 2 2) or even "Hello world!", compilation is most interesting in the context of procedures. Compiling a lambda expression produces a compiled procedure, which is just like a normal procedure except typically much faster, because it can bypass the generic interpreter.

Functions from system modules in a Guile installation are normally compiled already, so they load and run quickly.

Note that well-written Scheme programs will not typically call the procedures in this section, for the same reason that it is often bad taste to use eval. By default, Guile automatically compiles any files it encounters that have not been compiled yet (see Section 4.2 [Invoking Guile], page 35). The compiler can also be invoked explicitly from the shell as guild compile foo.scm.

(Why are calls to eval and compile usually in bad taste? Because they are limited, in that they can only really make sense for top-level expressions. Also, most needs for “compile-time” computation are fulfilled by macros and closures. Of course one good counterexample is the REPL itself, or any code that reads expressions from a port.)

Automatic compilation generally works transparently, without any need for user intervention. However Guile does not yet do proper dependency tracking, so that if file a.scm uses macros from b.scm, and b.scm changes, a.scm would not be automatically recompiled. To forcibly invalidate the auto-compilation cache, pass the --fresh-auto-compile option to Guile, or set the GUILE_AUTO_COMPILE environment variable to fresh (instead of to 0 or 1).

For more information on the compiler itself, see Section 9.4 [Compiling to the Virtual Machine], page 836. For information on the virtual machine, see Section 9.3 [A Virtual Machine for Guile], page 812.

The command-line interface to Guile's compiler is the guild compile command:

guild compile [option...] file...

Compile file, a source file, and store bytecode in the compilation cache or in the file specified by the -o option. The following options are available:

- -L dir
  --load-path=dir
  Add dir to the front of the module load path.
-o ofile
--output=ofile
Write output bytecode to ofile. By convention, bytecode file names end in .go. When -o is omitted, the output file name is as for compile-file (see below).

-x extension
Recognize extension as a valid source file name extension.
For example, to compile R6RS code, you might want to pass -x .sls so that files ending in .sls can be found.

-W warning
--warn=warning
Emit warnings of type warning; use --warn=help for a list of available warnings and their description. Currently recognized warnings include unused-variable, unused-toplevel, shadowed-toplevel, unbound-variable, macro-use-before-definition, arity-mismatch, format, duplicate-case-datum, and bad-case-datum.

-f lang
--from=lang
Use lang as the source language of file. If this option is omitted, scheme is assumed.

-t lang
--to=lang
Use lang as the target language of file. If this option is omitted, rtl is assumed.

-T target
--target=target
Produce code for target instead of %host-type (see Section 6.23.1 [Build Config], page 450). Target must be a valid GNU triplet, such as armv5tel-unknown-linux-gnueabi (see Section “Specifying Target Triplets” in GNU Autoconf Manual).

Each file is assumed to be UTF-8-encoded, unless it contains a coding declaration as recognized by file-encoding (see Section 6.18.8 [Character Encoding of Source Files], page 392).

The compiler can also be invoked directly by Scheme code using the procedures below:

\[
\text{compile } \text{exp} \left[ \#:\text{env}=#f \right] \left[ \#:\text{from}=(\text{current-language}) \right] \left[ \#:\text{to}=\text{value} \right] \left[ \#:\text{opts}=() \right] \left[ \text{Scheme Procedure} \right]
\]
Compile the expression exp in the environment env. If exp is a procedure, the result will be a compiled procedure; otherwise compile is mostly equivalent to eval.

For a discussion of languages and compiler options, See Section 9.4 [Compiling to the Virtual Machine], page 836.
compiled-file file [#:output-file=#f] [#:from=(current-language)] [#:to='rtl'] [#:env=(default-environment from)] [#:opts=''] [#:canonicalization='relative']

Compile the file named file.

Output will be written to a output-file. If you do not supply an output file name, output is written to a file in the cache directory, as computed by (compiled-file-name file).

from and to specify the source and target languages. See Section 9.4 [Compiling to the Virtual Machine], page 836, for more information on these options, and on env and opts.

As with guild compile, file is assumed to be UTF-8-encoded unless it contains a coding declaration.

compiled-file-name file

Compute a cached location for a compiled version of a Scheme file named file.

This file will usually be below the $HOME/.cache/guile/ccache directory, depending on the value of the XDG_CACHE_HOME environment variable. The intention is that compiled-file-name provides a fallback location for caching auto-compiled files. If you want to place a compile file in the %load-compiled-path, you should pass the output-file option to compile-file, explicitly.

%auto-compilation-options

This variable contains the options passed to the compile-file procedure when auto-compiling source files. By default, it enables useful compilation warnings. It can be customized from ~/.guile.

6.18.6 Loading Scheme Code from File

load filename [reader]

Load filename and evaluate its contents in the top-level environment.

reader if provided should be either #f, or a procedure with the signature (lambda (port) ...) which reads the next expression from port. If reader is #f or absent, Guile’s built-in read procedure is used (see Section 6.18.2 [Scheme Read], page 382).

The reader argument takes effect by setting the value of the current-reader fluid (see below) before loading the file, and restoring its previous value when loading is complete. The Scheme code inside filename can itself change the current reader procedure on the fly by setting current-reader fluid.

If the variable %load-hook is defined, it should be bound to a procedure that will be called before any code is loaded. See documentation for %load-hook later in this section.

load-compiled filename

Load the compiled file named filename.

Compiling a source file (see Section 6.18 [Read/Load/Eval/Compile], page 379) and then calling load-compiled on the resulting file is equivalent to calling load on the source file.
primitive-load filename

scm_primitive_load (filename)

Load the file named filename and evaluate its contents in the top-level environment. filename must either be a full pathname or be a pathname relative to the current directory. If the variable %load-hook is defined, it should be bound to a procedure that will be called before any code is loaded. See the documentation for %load-hook later in this section.

SCM scm_c_primitive_load (const char *filename)

scm_primitive_load, but taking a C string instead of an SCM.

current-reader

current-reader holds the read procedure that is currently being used by the above loading procedures to read expressions (from the file that they are loading). current-reader is a fluid, so it has an independent value in each dynamic root and should be read and set using fluid-ref and fluid-set! (see Section 6.13.11 [Fluids and Dynamic States], page 321).

Changing current-reader is typically useful to introduce local syntactic changes, such that code following the fluid-set! call is read using the newly installed reader. The current-reader change should take place at evaluation time when the code is evaluated, or at compilation time when the code is compiled:

(eval-when (compile eval)
 (fluid-set! current-reader my-own-reader))

The eval-when form above ensures that the current-reader change occurs at the right time.

%load-hook

A procedure to be called (%load-hook filename) whenever a file is loaded, or #f for no such call. %load-hook is used by all of the loading functions (load and primitive-load, and load-from-path and primitive-load-path documented in the next section).

For example an application can set this to show what’s loaded,

(set! %load-hook (lambda (filename)
 (format #t "Loading ~a ...
" filename)))

(load-from-path "foo.scm")

" Loading /usr/local/share/guile/site/foo.scm ...

current-load-port

Return the current-load-port. The load port is used internally by primitive-load.

6.18.7 Load Paths

The procedure in the previous section look for Scheme code in the file system at specific location. Guile also has some procedures to search the load path for code.

%load-path

List of directories which should be searched for Scheme modules and libraries. When Guile starts up, %load-path is initialized to the default load path (list
The GUILE_LOAD_PATH environment variable can be used to prepend or append additional directories (see Section 4.2.2 [Environment Variables], page 38).

See Section 6.23.1 [Build Config], page 450, for more on %site-dir and related procedures.

\texttt{load-from-path filename} \quad \textbf{[Scheme Procedure]}

Similar to \texttt{load}, but searches for \texttt{filename} in the load paths. Preferentially loads a compiled version of the file, if it is available and up-to-date.

A user can extend the load path by calling \texttt{add-to-load-path}.

\texttt{add-to-load-path dir} \quad \textbf{[Scheme Syntax]}

Add \texttt{dir} to the load path.

For example, a script might include this form to add the directory that it is in to the load path:

\begin{verbatim}
  (add-to-load-path (dirname (current-filename)))
\end{verbatim}

It’s better to use \texttt{add-to-load-path} than to modify \%load-path directly, because \texttt{add-to-load-path} takes care of modifying the path both at compile-time and at run-time.

\texttt{primitive-load-path filename [exception-on-not-found]} \quad \textbf{[Scheme Procedure]}

\texttt{scm_primitive_load_path (filename)} \quad \textbf{[C Function]}

Search \%load-path for the file named \texttt{filename} and load it into the top-level environment. If \texttt{filename} is a relative pathname and is not found in the list of search paths, an error is signalled. Preferentially loads a compiled version of the file, if it is available and up-to-date.

If \texttt{filename} is a relative pathname and is not found in the list of search paths, one of three things may happen, depending on the optional second argument, \texttt{exception-on-not-found}. If it is \#f, \#f will be returned. If it is a procedure, it will be called with no arguments. (This allows a distinction to be made between exceptions raised by loading a file, and exceptions related to the loader itself.) Otherwise an error is signalled.

For compatibility with Guile 1.8 and earlier, the C function takes only one argument, which can be either a string (the file name) or an argument list.

\texttt{%search-load-path filename} \quad \textbf{[Scheme Procedure]}

\texttt{scm_sys_search_load_path (filename)} \quad \textbf{[C Function]}

Search \%load-path for the file named \texttt{filename}, which must be readable by the current user. If \texttt{filename} is found in the list of paths to search or is an absolute pathname, return its full pathname. Otherwise, return \#f. Filenames may have any of the optional extensions in the \%load-extensions list; \%search-load-path will try each extension automatically.

\texttt{%load-extensions} \quad \textbf{[Variable]}

A list of default file extensions for files containing Scheme code. \%search-load-path tries each of these extensions when looking for a file to load. By default, \%load-extensions is bound to the list ("" ".scm").
As mentioned above, when Guile searches the %load-path for a source file, it will also search the %load-compiled-path for a corresponding compiled file. If the compiled file is as new or newer than the source file, it will be loaded instead of the source file, using load-compiled.

%load-compiled-path
[Variable]
Like %load-path, but for compiled files. By default, this path has two entries: one for compiled files from Guile itself, and one for site packages. The GUILE_LOAD_COMPILED_PATH environment variable can be used to prepend or append additional directories (see Section 4.2.2 [Environment Variables], page 38).

When primitive-load-path searches the %load-compiled-path for a corresponding compiled file for a relative path it does so by appending .go to the relative path. For example, searching for ice-9/popen could find /usr/lib/guile/2.2/ccache/ice-9/popen.go, and use it instead of /usr/share/guile/2.2/ice-9/popen.scm.

If primitive-load-path does not find a corresponding .go file in the %load-compiled-path, or the .go file is out of date, it will search for a corresponding auto-compiled file in the fallback path, possibly creating one if one does not exist.

See Section 4.7 [Installing Site Packages], page 57, for more on how to correctly install site packages. See Section 6.20.4 [Modules and the File System], page 412, for more on the relationship between load paths and modules. See Section 6.18.5 [Compilation], page 386, for more on the fallback path and auto-compilation.

Finally, there are a couple of helper procedures for general path manipulation.

parse-path path [tail]  [Scheme Procedure]
scm_parse_path (path, tail)  [C Function]
 Parse path, which is expected to be a colon-separated string, into a list and return the resulting list with tail appended. If path is #f, tail is returned.

parse-path-with-ellipsis path base  [Scheme Procedure]
scm_parse_path_with_ellipsis (path, base)  [C Function]
 Parse path, which is expected to be a colon-separated string, into a list and return the resulting list with base (a list) spliced in place of the ... path component, if present, or else base is added to the end. If path is #f, base is returned.

search-path path filename [extensions [require-exts?]]  [Scheme Procedure]
scm_search_path (path, filename, rest)  [C Function]
 Search path for a directory containing a file named filename. The file must be readable, and not a directory. If we find one, return its full filename; otherwise, return #f. If filename is absolute, return it unchanged. If given, extensions is a list of strings; for each directory in path, we search for filename concatenated with each extension. If require-exts? is true, require that the returned file name have one of the given extensions; if require-exts? is not given, it defaults to #f.

For compatibility with Guile 1.8 and earlier, the C function takes only three arguments.
6.18.8 Character Encoding of Source Files

Scheme source code files are usually encoded in ASCII or UTF-8, but the built-in reader can interpret other character encodings as well. When Guile loads Scheme source code, it uses the file-encoding procedure (described below) to try to guess the encoding of the file. In the absence of any hints, UTF-8 is assumed. One way to provide a hint about the encoding of a source file is to place a coding declaration in the top 500 characters of the file.

A coding declaration has the form coding: XXXXXX, where XXXXXX is the name of a character encoding in which the source code file has been encoded. The coding declaration must appear in a scheme comment. It can either be a semicolon-initiated comment, or the first block #! comment in the file.

The name of the character encoding in the coding declaration is typically lower case and containing only letters, numbers, and hyphens, as recognized by set-port-encoding! (see Section 6.14.1 [Ports], page 328). Common examples of character encoding names are utf-8 and iso-8859-1, as defined by IANA (http://www.iana.org/assignments/character-sets). Thus, the coding declaration is mostly compatible with Emacs.

However, there are some differences in encoding names recognized by Emacs and encoding names defined by IANA, the latter being essentially a subset of the former. For instance, latin-1 is a valid encoding name for Emacs, but it’s not according to the IANA standard, which Guile follows; instead, you should use iso-8859-1, which is both understood by Emacs and dubbed by IANA (IANA writes it uppercase but Emacs wants it lowercase and Guile is case insensitive.)

For source code, only a subset of all possible character encodings can be interpreted by the built-in source code reader. Only those character encodings in which ASCII text appears unmodified can be used. This includes UTF-8 and ISO-8859-1 through ISO-8859-15. The multi-byte character encodings UTF-16 and UTF-32 may not be used because they are not compatible with ASCII.

There might be a scenario in which one would want to read non-ASCII code from a port, such as with the function read, instead of with load. If the port’s character encoding is the same as the encoding of the code to be read by the port, not other special handling is necessary. The port will automatically do the character encoding conversion. The functions setlocale or by set-port-encoding! are used to set port encodings (see Section 6.14.1 [Ports], page 328).

If a port is used to read code of unknown character encoding, it can accomplish this in three steps. First, the character encoding of the port should be set to ISO-8859-1 using set-port-encoding!. Then, the procedure file-encoding, described below, is used to scan for a coding declaration when reading from the port. As a side effect, it rewinds the port after its scan is complete. After that, the port’s character encoding should be set to the encoding returned by file-encoding, if any, again by using set-port-encoding!. Then the code can be read as normal.

Alternatively, one can use the #:guess-encoding keyword argument of open-file and related procedures. See Section 6.14.10.1 [File Ports], page 341.
file-encoding port  

scm_file_encoding (port)  

Attempt to scan the first few hundred bytes from the port for hints about its character encoding. Return a string containing the encoding name or #f if the encoding cannot be determined. The port is rewound.

Currently, the only supported method is to look for an Emacs-like character coding declaration (see Section “Recognize Coding” in The GNU Emacs Reference Manual). The coding declaration is of the form coding: XXXX and must appear in a Scheme comment. Additional heuristics may be added in the future.

6.18.9 Delayed Evaluation

Promises are a convenient way to defer a calculation until its result is actually needed, and to run such a calculation only once. Also see Section 7.5.31 [SRFI-45], page 639.

delay expr  

Return a promise object which holds the given expr expression, ready to be evaluated by a later force.

promise? obj  

scm_promise_p (obj)  

Return true if obj is a promise.

force p  

scm_force (p)  

Return the value obtained from evaluating the expr in the given promise p. If p has previously been forced then its expr is not evaluated again, instead the value obtained at that time is simply returned.

During a force, an expr can call force again on its own promise, resulting in a recursive evaluation of that expr. The first evaluation to return gives the value for the promise. Higher evaluations run to completion in the normal way, but their results are ignored, force always returns the first value.

6.18.10 Local Evaluation

Guile includes a facility to capture a lexical environment, and later evaluate a new expression within that environment. This code is implemented in a module.

(use-modules (ice-9 local-eval))

the-environment  

Captures and returns a lexical environment for use with local-eval or local-compile.

local-eval exp env  

scm_local_eval (exp, env)  

local-compile exp env [opts=()]  

Evaluate or compile the expression exp in the lexical environment env.

Here is a simple example, illustrating that it is the variable that gets captured, not just its value at one point in time.

(define e (let ((x 100)) (the-environment)))
(define fetch-x (local-eval '(lambda () x) e))
(fetch-x)
⇒ 100
(local-eval '(set! x 42) e)
(fetch-x)
⇒ 42

While exp is evaluated within the lexical environment of (the-environment), it has the dynamic environment of the call to local-eval.

local-eval and local-compile can only evaluate expressions, not definitions.

(local-eval '(define foo 42)
  (let ((x 100)) (the-environment)))
⇒ syntax error: definition in expression context

Note that the current implementation of (the-environment) only captures “normal” lexical bindings, and pattern variables bound by syntax-case. It does not currently capture local syntax transformers bound by let-syntax, letrec-syntax or non-top-level define-syntax forms. Any attempt to reference such captured syntactic keywords via local-eval or local-compile produces an error.

6.18.11 Local Inclusion

This section has discussed various means of linking Scheme code together: fundamentally, loading up files at run-time using load and load-compiled. Guile provides another option to compose parts of programs together at expansion-time instead of at run-time.

include file-name

Open file-name, at expansion-time, and read the Scheme forms that it contains, splicing them into the location of the include, within a begin.

If file-name is a relative path, it is searched for relative to the path that contains the file that the include form appears in.

If you are a C programmer, if load in Scheme is like dlopen in C, consider include to be like the C preprocessor’s #include. When you use include, it is as if the contents of the included file were typed in instead of the include form.

Because the code is included at compile-time, it is available to the macroexpander. Syntax definitions in the included file are available to later code in the form in which the include appears, without the need for eval-when. (See Section 6.10.8 [Eval When], page 279.)

For the same reason, compiling a form that uses include results in one compilation unit, composed of multiple files. Loading the compiled file is one stat operation for the compilation unit, instead of 2*n in the case of load (once for each loaded source file, and once each corresponding compiled file, in the best case).

Unlike load, include also works within nested lexical contexts. It so happens that the optimizer works best within a lexical context, because all of the uses of bindings in a lexical context are visible, so composing files by including them within a (let () . . .) can sometimes lead to important speed improvements.
On the other hand, **include** does have all the disadvantages of early binding: once the code with the **include** is compiled, no change to the included file is reflected in the future behavior of the including form.

Also, the particular form of **include**, which requires an absolute path, or a path relative to the current directory at compile-time, is not very amenable to compiling the source in one place, but then installing the source to another place. For this reason, Guile provides another form, **include-from-path**, which looks for the source file to include within a load path.

```
include-from-path file-name
```

Like **include**, but instead of expecting **file-name** to be an absolute file name, it is expected to be a relative path to search in the `%load-path`.

**include-from-path** is more useful when you want to install all of the source files for a package (as you should!). It makes it possible to evaluate an installed file from source, instead of relying on the .go file being up to date.

### 6.18.12 Sandboxed Evaluation

Sometimes you would like to evaluate code that comes from an untrusted party. The safest way to do this is to buy a new computer, evaluate the code on that computer, then throw the machine away. However if you are unwilling to take this simple approach, Guile does include a limited “sandbox” facility that can allow untrusted code to be evaluated with some confidence.

To use the sandboxed evaluator, load its module:

```
(use-modules (ice-9 sandbox))
```

Guile’s sandboxing facility starts with the ability to restrict the time and space used by a piece of code.

```
call-with-time-limit limit thunk limit-reached
```

Call **thunk**, but cancel it if **limit** seconds of wall-clock time have elapsed. If the computation is cancelled, call **limit-reached** in tail position. **thunk** must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

```
call-with-allocation-limit limit thunk limit-reached
```

Call **thunk**, but cancel it if **limit** bytes have been allocated. If the computation is cancelled, call **limit-reached** in tail position. **thunk** must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

This limit applies to both stack and heap allocation. The computation will not be aborted before **limit** bytes have been allocated, but for the heap allocation limit, the check may be postponed until the next garbage collection.

Note that as a current shortcoming, the heap size limit applies to all threads; concurrent allocation by other unrelated threads counts towards the allocation limit.

```
call-with-time-and-allocation-limits time-limit allocation-limit thunk
```

Invoke **thunk** in a dynamic extent in which its execution is limited to **time-limit** seconds of wall-clock time, and its allocation to **allocation-limit** bytes. **thunk** must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.
If successful, return all values produced by invoking thunk. Any uncaught exception thrown by the thunk will propagate out. If the time or allocation limit is exceeded, an exception will be thrown to the limit-exceeded key.

The time limit and stack limit are both very precise, but the heap limit only gets checked asynchronously, after a garbage collection. In particular, if the heap is already very large, the number of allocated bytes between garbage collections will be large, and therefore the precision of the check is reduced.

Additionally, due to the mechanism used by the allocation limit (the after-gc-hook), large single allocations like (make-vector #e1e7) are only detected after the allocation completes, even if the allocation itself causes garbage collection. It’s possible therefore for user code to not only exceed the allocation limit set, but also to exhaust all available memory, causing out-of-memory conditions at any allocation site. Failure to allocate memory in Guile itself should be safe and cause an exception to be thrown, but most systems are not designed to handle malloc failures. An allocation failure may therefore exercise unexpected code paths in your system, so it is a weakness of the sandbox (and therefore an interesting point of attack).

The main sandbox interface is eval-in-sandbox.

eval-in-sandbox exp [#:time-limit 0.1] [#:allocation-limit #e10e6] [#:bindings all-pure-bindings] [#:module (make-sandbox-module bindings)] [#:sever-module? #t]
Evaluate the Scheme expression exp within an isolated "sandbox". Limit its execution to time-limit seconds of wall-clock time, and limit its allocation to allocation-limit bytes.

The evaluation will occur in module, which defaults to the result of calling make-sandbox-module on bindings, which itself defaults to all-pure-bindings. This is the core of the sandbox: creating a scope for the expression that is safe.

A safe sandbox module has two characteristics. Firstly, it will not allow the expression being evaluated to avoid being cancelled due to time or allocation limits. This ensures that the expression terminates in a timely fashion.

Secondly, a safe sandbox module will prevent the evaluation from receiving information from previous evaluations, or from affecting future evaluations. All combinations of binding sets exported by (ice-9 sandbox) form safe sandbox modules.

The bindings should be given as a list of import sets. One import set is a list whose car names an interface, like (ice-9 q), and whose cdr is a list of imports. An import is either a bare symbol or a pair of (out . in), where out and in are both symbols and denote the name under which a binding is exported from the module, and the name under which to make the binding available, respectively. Note that bindings is only used as an input to the default initializer for the module argument; if you pass #:module, bindings is unused. If sever-module? is true (the default), the module will be unlinked from the global module tree after the evaluation returns, to allow mod to be garbage-collected.

If successful, return all values produced by exp. Any uncaught exception thrown by the expression will propagate out. If the time or allocation limit is exceeded, an exception will be thrown to the limit-exceeded key.
Constructing a safe sandbox module is tricky in general. Guile defines an easy way to construct safe modules from predefined sets of bindings. Before getting to that interface, here are some general notes on safety.

1. The time and allocation limits rely on the ability to interrupt and cancel a computation. For this reason, no binding included in a sandbox module should be able to indefinitely postpone interrupt handling, nor should a binding be able to prevent an abort. In practice this second consideration means that `dynamic-wind` should not be included in any binding set.

2. The time and allocation limits apply only to the `eval-in-sandbox` call. If the call returns a procedure which is later called, no limit is “automatically” in place. Users of `eval-in-sandbox` have to be very careful to reimpose limits when calling procedures that escape from sandboxes.

3. Similarly, the dynamic environment of the `eval-in-sandbox` call is not necessarily in place when any procedure that escapes from the sandbox is later called. This detail prevents us from exposing `primitive-eval` to the sandbox, for two reasons. The first is that it’s possible for legacy code to forge references to any binding, if the `allow-legacy-syntax-objects?` parameter is true. The default for this parameter is true; see Section 6.10.4 [Syntax Transformer Helpers], page 272, for the details. The parameter is bound to `#f` for the duration of the `eval-in-sandbox` call itself, but that will not be in place during calls to escaped procedures. The second reason we don’t expose `primitive-eval` is that `primitive-eval` implicitly works in the current module, which for an escaped procedure will probably be different than the module that is current for the `eval-in-sandbox` call itself. The common denominator here is that if an interface exposed to the sandbox relies on dynamic environments, it is easy to mistakenly grant the sandboxed procedure additional capabilities in the form of bindings that it should not have access to. For this reason, the default sets of predefined bindings do not depend on any dynamically scoped value.

4. Mutation may allow a sandboxed evaluation to break some invariant in users of data supplied to it. A lot of code culturally doesn’t expect mutation, but if you hand mutable data to a sandboxed evaluation and you also grant mutating capabilities to that evaluation, then the sandboxed code may indeed mutate that data. The default set of bindings to the sandbox do not include any mutating primitives. Relatedly, `set!` may allow a sandbox to mutate a primitive, invalidating many system-wide invariants. Guile is currently quite permissive when it comes to imported bindings and mutability. Although `set!` to a module-local or lexically bound variable would be fine, we don’t currently have an easy way to disallow `set!` to an imported binding, so currently no binding set includes `set!`.

5. Mutation may allow a sandboxed evaluation to keep state, or make a communication mechanism with other code. On the one hand this sounds cool, but on the other hand maybe this is part of your threat model. Again, the default set of bindings doesn’t include mutating primitives, preventing sandboxed evaluations from keeping state.

6. The sandbox should probably not be able to open a network connection, or write to a file, or open a file from disk. The default binding set includes no interaction with the operating system.
If you, dear reader, find the above discussion interesting, you will enjoy Jonathan Rees’ dissertation, “A Security Kernel Based on the Lambda Calculus”.

**all-pure-bindings**  
All “pure” bindings that together form a safe subset of those bindings available by default to Guile user code.

**all-pure-and-impure-bindings**  
Like **all-pure-bindings**, but additionally including mutating primitives like `vector-set!`. This set is still safe in the sense mentioned above, with the caveats about mutation.

The components of these composite sets are as follows:

*alist-bindings*  
array-bindings  
bit-bindings  
bitvector-bindings  
char-bindings  
char-set-bindings  
clock-bindings  
core-bindings  
error-bindings  
fluid-bindings  
hash-bindings  
iteration-bindings  
keyword-bindings  
list-bindings  
macro-bindings  
nil-bindings  
number-bindings  
pair-bindings  
predicate-bindings  
procedure-bindings  
promise-bindings  
prompt-bindings  
regexp-bindings  
sort-bindings  
srfi-4-bindings  
string-bindings  
symbol-bindings  
unspecified-bindings  
variable-bindings  
vector-bindings  
version-bindings

The components of **all-pure-bindings**.

*mutating-alist-bindings*  
mutating-array-bindings
The additional components of `all-pure-and-impure-bindings`.

Finally, what do you do with a binding set? What is a binding set anyway? `make-sandbox-module` is here for you.

```
make-sandbox-module bindings
```

Return a fresh module that only contains `bindings`.

The `bindings` should be given as a list of import sets. One import set is a list whose car names an interface, like `(ice-9 q)`, and whose cdr is a list of imports. An import is either a bare symbol or a pair of `(out . in)`, where `out` and `in` are both symbols and denote the name under which a binding is exported from the module, and the name under which to make the binding available, respectively.

So you see that binding sets are just lists, and `all-pure-and-impure-bindings` is really just the result of appending all of the component binding sets.

### 6.18.13 REPL Servers

The procedures in this section are provided by

```
(use-modules (system repl server))
```

When an application is written in Guile, it is often convenient to allow the user to be able to interact with it by evaluating Scheme expressions in a REPL.

The procedures of this module allow you to spawn a `REPL server`, which permits interaction over a local or TCP connection. Guile itself uses them internally to implement the `--listen` switch, Section 4.2.1 [Command-line Options], page 35.

```
make-tcp-server-socket [#:host=#f] [#:addr] [#:port=37146]
```

Return a stream socket bound to a given address `addr` and port number `port`. If the `host` is given, and `addr` is not, then the `host` string is converted to an address. If neither is given, we use the loopback address.

```
make-unix-domain-server-socket
[#:path="/tmp/guile-socket"]
```

Return a UNIX domain socket, bound to a given `path`. 
run-server [server-socket]  [Scheme Procedure]
spawm-server [server-socket]  [Scheme Procedure]
Create and run a REPL, making it available over the given server-socket. If server-socket is not provided, it defaults to the socket created by calling make-tcp-server-socket with no arguments.
runt-server runs the server in the current thread, whereas spawn-server runs the server in a new thread.

stop-server-and-clients!  [Scheme Procedure]
Closes the connection on all running server sockets.
Please note that in the current implementation, the REPL threads are cancelled without unwinding their stacks. If any of them are holding mutexes or are within a critical section, the results are unspecified.

6.18.14 Cooperative REPL Servers
The procedures in this section are provided by

(use-modules (system repl coop-server))

Whereas ordinary REPL servers run in their own threads (see Section 6.18.13 [REPL Servers], page 399), sometimes it is more convenient to provide REPLs that run at specified times within an existing thread, for example in programs utilizing an event loop or in single-threaded programs. This allows for safe access and mutation of a program’s data structures from the REPL, without concern for thread synchronization.

Although the REPLs are run in the thread that calls spawn-coop-repl-server and poll-coop-repl-server, dedicated threads are spawned so that the calling thread is not blocked. The spawned threads read input for the REPLs and to listen for new connections.

Cooperative REPL servers must be polled periodically to evaluate any pending expressions by calling poll-coop-repl-server with the object returned from spawn-coop-repl-server. The thread that calls poll-coop-repl-server will be blocked for as long as the expression takes to be evaluated or if the debugger is entered.

spawn-coop-repl-server [server-socket]  [Scheme Procedure]
Create and return a new cooperative REPL server object, and spawn a new thread to listen for connections on server-socket. Proper functioning of the REPL server requires that poll-coop-repl-server be called periodically on the returned server object.

poll-coop-repl-server coop-server  [Scheme Procedure]
Poll the cooperative REPL server coop-server and apply a pending operation if there is one, such as evaluating an expression typed at the REPL prompt. This procedure must be called from the same thread that called spawn-coop-repl-server.

6.19 Memory Management and Garbage Collection
Guile uses a garbage collector to manage most of its objects. While the garbage collector is designed to be mostly invisible, you sometimes need to interact with it explicitly.

See Section 5.4.2 [Garbage Collection], page 67, for a general discussion of how garbage collection relates to using Guile from C.
6.19.1 Function related to Garbage Collection

**gc**

[Scheme Procedure]

Finds all of the “live” SCM objects and reclaims for further use those that are no longer accessible. You normally don’t need to call this function explicitly. Its functionality is invoked automatically as needed.

**scm_gc**

[C Function]

Finds all of the “live” SCM objects and reclaims for further use those that are no longer accessible. You normally don’t need to call this function explicitly. Its functionality is invoked automatically as needed.

**SCM scm_gc_protect_object (SCM obj)**

[C Function]

Protects obj from being freed by the garbage collector, when it otherwise might be. When you are done with the object, call **scm_gc_unprotect_object** on the object. Calls to **scm_gc_protect_object**/**scm_gc_unprotect_object** can be nested, and the object remains protected until it has been unprotected as many times as it was protected. It is an error to unprotected an object more times than it has been protected. Returns the SCM object it was passed.

Note that storing obj in a C global variable has the same effect.9

**SCM scm_gc_unprotect_object (SCM obj)**

[C Function]

Unprotects an object from the garbage collector which was protected by **scm_gc_unprotect_object**. Returns the SCM object it was passed.

**SCM scm_permanent_object (SCM obj)**

[C Function]

Similar to **scm_gc_protect_object** in that it causes the collector to always mark the object, except that it should not be nested (only call **scm_permanent_object** on an object once), and it has no corresponding unpermanent function. Once an object is declared permanent, it will never be freed. Returns the SCM object it was passed.

**void scm_remember_upto_here_1 (SCM obj)**

[C Macro]

Create a reference to the given object or objects, so they’re certain to be present on the stack or in a register and hence will not be freed by the garbage collector before this point.

Note that these functions can only be applied to ordinary C local variables (ie. “automatics”). Objects held in global or static variables or some malloced block or the like cannot be protected with this mechanism.

**gc-stats**

[Scheme Procedure]

**scm_gc_stats**

[C Function]

Return an association list of statistics about Guile’s current use of storage.

**gc-live-object-stats**

[Scheme Procedure]

**scm_gc_live_object_stats**

[C Function]

Return an alist of statistics of the current live objects.

**void scm_gc_mark (SCM x)**

[Function]

Mark the object x, and recurse on any objects x refers to. If x’s mark bit is already set, return immediately. This function must only be called during the mark-phase of garbage collection, typically from a smob **mark** function.

---

9 In Guile up to version 1.8, C global variables were not visited by the garbage collector in the mark phase; hence, **scm_gc_protect_object** was the only way in C to prevent a Scheme object from being freed.
6.19.2 Memory Blocks

In C programs, dynamic management of memory blocks is normally done with the functions `malloc`, `realloc`, and `free`. Guile has additional functions for dynamic memory allocation that are integrated into the garbage collector and the error reporting system.

Memory blocks that are associated with Scheme objects (for example a foreign object) should be allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless`. These two functions will either return a valid pointer or signal an error. Memory blocks allocated this way may be released explicitly; however, this is not strictly needed, and we recommend not calling `scm_gc_free`. All memory allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless` is automatically reclaimed when the garbage collector no longer sees any live reference to it\(^\text{10}\).

When garbage collection occurs, Guile will visit the words in memory allocated with `scm_gc_malloc`, looking for live pointers. This means that if `scm_gc_malloc`-allocated memory contains a pointer to some other part of the memory, the garbage collector notices it and prevents it from being reclaimed\(^\text{11}\). Conversely, memory allocated with `scm_gc_malloc_pointerless` is assumed to be “pointer-less” and is not scanned for pointers.

For memory that is not associated with a Scheme object, you can use `scm_malloc` instead of `malloc`. Like `scm_gc_malloc`, it will either return a valid pointer or signal an error. However, it will not assume that the new memory block can be freed by a garbage collection. The memory must be explicitly freed with `free`.

There is also `scm_gc_realloc` and `scm_realloc`, to be used in place of `realloc` when appropriate, and `scm_gc_calloc` and `scm_calloc`, to be used in place of `calloc` when appropriate.

The function `scm_dynwind_free` can be useful when memory should be freed with libc’s `free` when leaving a dynwind context, See Section 6.13.10 [Dynamic Wind], page 317.

\begin{verbatim}
void * scm_malloc (size_t size)  [C Function]
void * scm calloc (size_t size)  [C Function]
Allocate size bytes of memory and return a pointer to it. When size is 0, return NULL.
When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.
The memory is allocated by the libc malloc function and can be freed with free.
There is no scm_free function to go with scm_malloc to make it easier to pass memory back and forth between different modules.
The function scm calloc is similar to scm malloc, but initializes the block of memory
to zero as well.
These functions will (indirectly) call scm_gc_register_allocation.
void * scm realloc (void *mem, size_t new_size)  [C Function]
Change the size of the memory block at mem to new_size and return its new location.
When new_size is 0, this is the same as calling free on mem and NULL is returned.
\end{verbatim}

\(^{10}\) In Guile up to version 1.8, memory allocated with `scm_gc_malloc` had to be freed with `scm_gc_free`.
\(^{11}\) In Guile up to 1.8, memory allocated with `scm_gc_malloc` was not visited by the collector in the mark phase. Consequently, the GC had to be told explicitly about pointers to live objects contained in the memory block, e.g., via SMOB mark functions (see Section 6.8 [Smobs], page 245)
When \texttt{mem} is \texttt{NULL}, this function behaves like \texttt{scm_malloc} and allocates a new block of size \texttt{new_size}.

When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

This function will call \texttt{scm_gc_register_allocation}.

\begin{verbatim}
void * scm_gc_malloc (size_t size, const char *what) [C Function]
void * scm_gc_malloc_pointerless (size_t size, const char *what) [C Function]
void * scm_gc_realloc (void *mem, size_t old_size, size_t new_size, const char *what); [C Function]
void * scm_gc_calloc (size_t size, const char *what) [C Function]
\end{verbatim}

Allocate \texttt{size} bytes of automatically-managed memory. The memory is automatically freed when no longer referenced from any live memory block.

When garbage collection occurs, Guile will visit the words in memory allocated with \texttt{scm_gc_malloc} or \texttt{scm_gc_calloc}, looking for pointers to other memory allocations that are managed by the GC. In contrast, memory allocated by \texttt{scm_gc_malloc_pointerless} is not scanned for pointers.

The \texttt{scm_gc_realloc} call preserves the “pointerlessness” of the memory area pointed to by \texttt{mem}. Note that you need to pass the old size of a reallocated memory block as well. See below for a motivation.

\begin{verbatim}
void scm_gc_free (void *mem, size_t size, const char *what) [C Function]
\end{verbatim}

Explicitly free the memory block pointed to by \texttt{mem}, which was previously allocated by one of the above \texttt{scm_gc} functions. This function is almost always unnecessary, except for codebases that still need to compile on Guile 1.8.

Note that you need to explicitly pass the \texttt{size} parameter. This is done since it should normally be easy to provide this parameter (for memory that is associated with GC controlled objects) and help keep the memory management overhead very low. However, in Guile 2.x, \texttt{size} is always ignored.

\begin{verbatim}
void scm_gc_register_allocation (size_t size) [C Function]
\end{verbatim}

Informs the garbage collector that \texttt{size} bytes have been allocated, which the collector would otherwise not have known about.

In general, Scheme will decide to collect garbage only after some amount of memory has been allocated. Calling this function will make the Scheme garbage collector know about more allocation, and thus run more often (as appropriate).

It is especially important to call this function when large unmanaged allocations, like images, may be freed by small Scheme allocations, like foreign objects.

\begin{verbatim}
void scm_dynwind_free (void *mem) [C Function]
\end{verbatim}

Equivalent to \texttt{scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)}. That is, the memory block at \texttt{mem} will be freed (using \texttt{free} from the C library) when the current dynwind is left.
**malloc-stats**

Return an alist ((what . n) ...) describing number of malloced objects. what is the second argument to scm_gc_malloc, n is the number of objects of that type currently allocated.

This function is only available if the GUILE_DEBUG_MALLOC preprocessor macro was defined when Guile was compiled.

### 6.19.3 Weak References

[FIXME: This chapter is based on Mikael Djurfeldt’s answer to a question by Michael Livshin. Any mistakes are not theirs, of course. ]

Weak references let you attach bookkeeping information to data so that the additional information automatically disappears when the original data is no longer in use and gets garbage collected. In a weak key hash, the hash entry for that key disappears as soon as the key is no longer referenced from anywhere else. For weak value hashes, the same happens as soon as the value is no longer in use. Entries in a doubly weak hash disappear when either the key or the value are not used anywhere else anymore.

Object properties offer the same kind of functionality as weak key hashes in many situations. (see Section 6.11.2 [Object Properties], page 285)

Here’s an example (a little bit strained perhaps, but one of the examples is actually used in Guile):

Assume that you’re implementing a debugging system where you want to associate information about filename and position of source code expressions with the expressions themselves.

Hashables can be used for that, but if you use ordinary hash tables it will be impossible for the scheme interpreter to “forget” old source when, for example, a file is reloaded.

To implement the mapping from source code expressions to positional information it is necessary to use weak-key tables since we don’t want the expressions to be remembered just because they are in our table.

To implement a mapping from source file line numbers to source code expressions you would use a weak-value table.

To implement a mapping from source code expressions to the procedures they constitute a doubly-weak table has to be used.

#### 6.19.3.1 Weak hash tables

- **make-weak-key-hash-table [size]**
  
  [Scheme Procedure]

- **make-weak-value-hash-table [size]**
  
  [Scheme Procedure]

- **make-doubly-weak-hash-table [size]**
  
  [Scheme Procedure]

- **scm_make_weak_key_hash_table (size)**
  
  [C Function]

- **scm_make_weak_value_hash_table (size)**
  
  [C Function]

- **scm_make_doubly_weak_hash_table (size)**
  
  [C Function]

  Return a weak hash table with size buckets. As with any hash table, choosing a good size for the table requires some caution.

  You can modify weak hash tables in exactly the same way you would modify regular hash tables, with the exception of the routines that act on handles. Weak tables
have a different implementation behind the scenes that doesn’t have handles. See Section 6.6.22 [Hash Tables], page 237, for more on hashq-ref et al.

Note that in a weak-key hash table, the reference to the value is strong. This means that if the value references the key, even indirectly, the key will never be collected, which can lead to a memory leak. The reverse is true for weak value tables.

weak-key-hash-table? obj
weak-value-hash-table? obj
doubly-weak-hash-table? obj
scm_weak_key_hash_table_p (obj)
scm_weak_value_hash_table_p (obj)
scm_doubly_weak_hash_table_p (obj)

Return #t if obj is the specified weak hash table. Note that a doubly weak hash table is neither a weak key nor a weak value hash table.

6.19.3.2 Weak vectors

make-weak-vector size [fill]
scm_make_weak_vector (size, fill)

Return a weak vector with size elements. If the optional argument fill is given, all entries in the vector will be set to fill. The default value for fill is the empty list.

weak-vector elem ...
list->weak-vector l
scm_weak_vector (l)

Construct a weak vector from a list: weak-vector uses the list of its arguments while list->weak-vector uses its only argument l (a list) to construct a weak vector the same way list->vector would.

weak-vector? obj
scm_weak_vector_p (obj)

Return #t if obj is a weak vector.

weak-vector-ref wvect k
scm_weak_vector_ref (wvect, k)

Return the kth element of the weak vector wvect, or #f if that element has been collected.

weak-vector-set! wvect k elt
scm_weak_vector_set_x (wvect, k, elt)

Set the kth element of the weak vector wvect to elt.

6.19.4 Guardians

Guardians provide a way to be notified about objects that would otherwise be collected as garbage. Guarding them prevents the objects from being collected and cleanup actions can be performed on them, for example.

make-guardian

scm_make_guardian ()

Create a new guardian. A guardian protects a set of objects from garbage collection, allowing a program to apply cleanup or other actions.

make-guardian returns a procedure representing the guardian. Calling the guardian procedure with an argument adds the argument to the guardian’s set of protected objects. Calling the guardian procedure without an argument returns one of the protected objects which are ready for garbage collection, or #f if no such object is available. Objects which are returned in this way are removed from the guardian.

You can put a single object into a guardian more than once and you can put a single object into more than one guardian. The object will then be returned multiple times by the guardian procedures.

An object is eligible to be returned from a guardian when it is no longer referenced from outside any guardian.

There is no guarantee about the order in which objects are returned from a guardian. If you want to impose an order on finalization actions, for example, you can do that by keeping objects alive in some global data structure until they are no longer needed for finalizing other objects.

Being an element in a weak vector, a key in a hash table with weak keys, or a value in a hash table with weak values does not prevent an object from being returned by a guardian. But as long as an object can be returned from a guardian it will not be removed from such a weak vector or hash table. In other words, a weak link does not prevent an object from being considered collectable, but being inside a guardian prevents a weak link from being broken.

A key in a weak key hash table can be thought of as having a strong reference to its associated value as long as the key is accessible. Consequently, when the key is only accessible from within a guardian, the reference from the key to the value is also considered to be coming from within a guardian. Thus, if there is no other reference to the value, it is eligible to be returned from a guardian.

6.20 Modules

When programs become large, naming conflicts can occur when a function or global variable defined in one file has the same name as a function or global variable in another file. Even just a similarity between function names can cause hard-to-find bugs, since a programmer might type the wrong function name.

The approach used to tackle this problem is called information encapsulation, which consists of packaging functional units into a given name space that is clearly separated from other name spaces.

The language features that allow this are usually called the module system because programs are broken up into modules that are compiled separately (or loaded separately in an interpreter).

Older languages, like C, have limited support for name space manipulation and protection. In C a variable or function is public by default, and can be made local to a module
with the `static` keyword. But you cannot reference public variables and functions from
another module with different names.

More advanced module systems have become a common feature in recently designed
languages: ML, Python, Perl, and Modula 3 all allow the renaming of objects from a
foreign module, so they will not clutter the global name space.

In addition, Guile offers variables as first-class objects. They can be used for interacting
with the module system.

### 6.20.1 General Information about Modules

A Guile module can be thought of as a collection of named procedures, variables and macros. More precisely, it is a set of bindings of symbols (names) to Scheme objects.

Within a module, all bindings are visible. Certain bindings can be declared public, in which case they are added to the module’s so-called export list; this set of public bindings is called the module’s public interface (see Section 6.20.3 [Creating Guile Modules], page 409).

A client module uses a providing module’s bindings by either accessing the providing module’s public interface, or by building a custom interface (and then accessing that). In a custom interface, the client module can select which bindings to access and can also algorithmically rename bindings. In contrast, when using the providing module’s public interface, the entire export list is available without renaming (see Section 6.20.2 [Using Guile Modules], page 407).

All Guile modules have a unique module name, for example `(ice-9 popen)` or `(srfi srfi-11)`. Module names are lists of one or more symbols.

When Guile goes to use an interface from a module, for example `(ice-9 popen)`, Guile first looks to see if it has loaded `(ice-9 popen)` for any reason. If the module has not been loaded yet, Guile searches a load path for a file that might define it, and loads that file.

The following subsections go into more detail on using, creating, installing, and otherwise manipulating modules and the module system.

### 6.20.2 Using Guile Modules

To use a Guile module is to access either its public interface or a custom interface (see Section 6.20.1 [General Information about Modules], page 407). Both types of access are handled by the syntactic form `use-modules`, which accepts one or more interface specifications and, upon evaluation, arranges for those interfaces to be available to the current module. This process may include locating and loading code for a given module if that code has not yet been loaded, following `%load-path` (see Section 6.20.4 [Modules and the File System], page 412).

An interface specification has one of two forms. The first variation is simply to name the module, in which case its public interface is the one accessed. For example:

```scheme
(use-modules (ice-9 popen))
```

Here, the interface specification is `(ice-9 popen)`, and the result is that the current module now has access to `open-pipe`, `close-pipe`, `open-input-pipe`, and so on (see Section 7.2.10 [Pipes], page 524).

Note in the previous example that if the current module had already defined `open-pipe`, that definition would be overwritten by the definition in `(ice-9 popen)`. For this reason
(and others), there is a second variation of interface specification that not only names a module to be accessed, but also selects bindings from it and renames them to suit the current module’s needs. For example:

```
(use-modules ((ice-9 popen)
   #:select ((open-pipe . pipe-open) close-pipe)
   #:renamer (symbol-prefix-proc 'unixy:)))
```

or more simply:

```
(use-modules ((ice-9 popen)
   #:select ((open-pipe . pipe-open) close-pipe)
   #:prefix unixy:))
```

Here, the interface specification is more complex than before, and the result is that a custom interface with only two bindings is created and subsequently accessed by the current module. The mapping of old to new names is as follows:

<table>
<thead>
<tr>
<th>(ice-9 popen) sees:</th>
<th>current module sees:</th>
</tr>
</thead>
<tbody>
<tr>
<td>open-pipe</td>
<td>unixy:pipe-open</td>
</tr>
<tr>
<td>close-pipe</td>
<td>unixy:close-pipe</td>
</tr>
</tbody>
</table>

This example also shows how to use the convenience procedure `symbol-prefix-proc`.

You can also directly refer to bindings in a module by using the @ syntax. For example, instead of using the `use-modules` statement from above and writing `unixy:pipe-open` to refer to the `pipe-open` from the `(ice-9 popen)`, you could also write (@ (ice-9 popen) open-pipe). Thus an alternative to the complete `use-modules` statement would be

```
(define unixy:pipe-open (@ (ice-9 popen) open-pipe))
(define unixy:close-pipe (@ (ice-9 popen) close-pipe))
```

There is also @@, which can be used like @, but does not check whether the variable that is being accessed is actually exported. Thus, @@ can be thought of as the impolite version of @ and should only be used as a last resort or for debugging, for example.

Note that just as with a `use-modules` statement, any module that has not yet been loaded will be loaded when referenced by a @ or @@ form.

You can also use the @ and @@ syntaxes as the target of a `set!` when the binding refers to a variable.

`symbol-prefix-proc prefix-sym`  
[Scheme Procedure]

Return a procedure that prefixes its arg (a symbol) with prefix-sym.

`use-modules spec ...`  
[syntax]

Resolve each interface specification spec into an interface and arrange for these to be accessible by the current module. The return value is unspecified.

`spec` can be a list of symbols, in which case it names a module whose public interface is found and used.

`spec` can also be of the form:

```
(MODULE-NAME [#:select SELECTION]
   [#:prefix PREFIX]
   [#:renamer RENAMER])
```

in which case a custom interface is newly created and used. `module-name` is a list of symbols, as above; `selection` is a list of selection-specs; `prefix` is a symbol that is
prepended to imported names; and \textit{renamer} is a procedure that takes a symbol and returns its new name. A selection-spec is either a symbol or a pair of symbols (\texttt{ORIG \ . \ SEEN}), where \textit{orig} is the name in the used module and \textit{seen} is the name in the using module. Note that \textit{seen} is also modified by \textit{prefix} and \textit{renamer}.

The \#:select, \#:prefix, and \#:renamer clauses are optional. If all are omitted, the returned interface has no bindings. If the \#:select clause is omitted, \textit{prefix} and \textit{renamer} operate on the used module’s public interface.

In addition to the above, \textit{spec} can also include a \#:version clause, of the form:

\texttt{#:version VERSION\textendash SPEC}

where \textit{version\textendash SPEC} is an R6RS-compatible version reference. An error will be signaled in the case in which a module with the same name has already been loaded, if that module specifies a version and that version is not compatible with \textit{version\textendash SPEC}. See Section 6.20.5 [R6RS Version References], page 413, for more on version references.

If the module name is not resolvable, \texttt{use-modules} will signal an error.

\texttt{@@ module-name binding-name} \hspace{1em} [syntax]

Refer to the binding named \textit{binding-name} in module \textit{module-name}. The binding must not have been exported by the module.

\texttt{@@ module-name binding-name} \hspace{1em} [syntax]

Refer to the binding named \textit{binding-name} in module \textit{module-name}. The binding must not have been exported by the module. This syntax is only intended for debugging purposes or as a last resort.

### 6.20.3 Creating Guile Modules

When you want to create your own modules, you have to take the following steps:

- Create a Scheme source file and add all variables and procedures you wish to export, or which are required by the exported procedures.
- Add a \texttt{define-module} form at the beginning.
- Export all bindings which should be in the public interface, either by using \texttt{define-public} or \texttt{export} (both documented below).

\texttt{define-module module-name option \ldots} \hspace{1em} [syntax]

\textit{module-name} is a list of one or more symbols.

\texttt{(define-module (ice-9 popen))}

\texttt{define-module} makes this module available to Guile programs under the given \textit{module-name}.

\textit{option} \ldots are keyword/value pairs which specify more about the defined module. The recognized options and their meaning are shown in the following table.

\texttt{#:use-module interface-specification}

Equivalent to a \texttt{(use-modules interface-specification)} (see Section 6.20.2 [Using Guile Modules], page 407).

\texttt{#:autoload module symbol-list}

Load \textit{module} when any of \textit{symbol-list} are accessed. For example,

\texttt{(define-module (my mod)}
#:autoload (srfi srfi-1) (partition delete-duplicates))
...
(if something
  (set! foo (delete-duplicates ...)))

When a module is autoloaded, all its bindings become available. symbol-list is just those that will first trigger the load.

An autoload is a good way to put off loading a big module until it’s really needed, for instance for faster startup or if it will only be needed in certain circumstances.

@ can do a similar thing (see Section 6.20.2 [Using Guile Modules], page 407), but in that case an @ form must be written every time a binding from the module is used.

#:export list

Export all identifiers in list which must be a list of symbols or pairs of symbols. This is equivalent to (export list) in the module body.

#:re-export list

Re-export all identifiers in list which must be a list of symbols or pairs of symbols. The symbols in list must be imported by the current module from other modules. This is equivalent to re-export below.

#:replace list

Export all identifiers in list (a list of symbols or pairs of symbols) and mark them as replacing bindings. In the module user’s name space, this will have the effect of replacing any binding with the same name that is not also “replacing”. Normally a replacement results in an “override” warning message, #:replace avoids that.

In general, a module that exports a binding for which the (guile) module already has a definition should use #:replace instead of #:export. #:replace, in a sense, lets Guile know that the module purposefully replaces a core binding. It is important to note, however, that this binding replacement is confined to the name space of the module user. In other words, the value of the core binding in question remains unchanged for other modules.

Note that although it is often a good idea for the replaced binding to remain compatible with a binding in (guile), to avoid surprising the user, sometimes the bindings will be incompatible. For example, SRFI-19 exports its own version of current-time (see Section 7.5.16.2 [SRFI-19 Time], page 609) which is not compatible with the core current-time function (see Section 7.2.5 [Time], page 509). Guile assumes that a user importing a module knows what she is doing, and uses #:replace for this binding rather than #:export.

A #:replace clause is equivalent to (export! list) in the module body.

The #:duplicates (see below) provides fine-grain control about duplicate binding handling on the module-user side.
#:version list
Specify a version for the module in the form of list, a list of zero or more
exact, nonnegative integers. The corresponding #:version option in the
use-modules form allows callers to restrict the value of this option in
various ways.

#:duplicates list
Tell Guile to handle duplicate bindings for the bindings imported by the
current module according to the policy defined by list, a list of symbols.
list must contain symbols representing a duplicate binding handling policy
chosen among the following:

check  Raises an error when a binding is imported from more than
       one place.

warn   Issue a warning when a binding is imported from more than
       one place and leave the responsibility of actually handling the
duplication to the next duplicate binding handler.

replace When a new binding is imported that has the same name as a
       previously imported binding, then do the following:
       1. If the old binding was said to be replacing (via the
           #:replace option above) and the new binding is not
           replacing, the keep the old binding.
       2. If the old binding was not said to be replacing and the
           new binding is replacing, then replace the old binding
           with the new one.
       3. If neither the old nor the new binding is replacing, then
           keep the old one.

warn-override-core Issue a warning when a core binding is being overwritten and
                        actually override the core binding with the new one.

first In case of duplicate bindings, the firstly imported binding is
       always the one which is kept.

last  In case of duplicate bindings, the lastly imported binding is
       always the one which is kept.

noop  In case of duplicate bindings, leave the responsibility to the
       next duplicate handler.

If list contains more than one symbol, then the duplicate binding handlers
which appear first will be used first when resolving a duplicate binding
situation. As mentioned above, some resolution policies may explicitly
leave the responsibility of handling the duplication to the next handler in list.

If GOOPS has been loaded before the #:duplicates clause is processed,
there are additional strategies available for dealing with generic functions.
See Section 8.6.3 [Merging Generics], page 768, for more information.
The default duplicate binding resolution policy is given by the `default-duplicate-binding-handler` procedure, and is

```
(replace warn-override-core warn last)
```

#:pure Create a pure module, that is a module which does not contain any of the standard procedure bindings except for the syntax forms. This is useful if you want to create safe modules, that is modules which do not know anything about dangerous procedures.

```scheme
(export variable ...)
```

Add all variables (which must be symbols or pairs of symbols) to the list of exported bindings of the current module. If `variable` is a pair, its car gives the name of the variable as seen by the current module and its cdr specifies a name for the binding in the current module’s public interface.

```scheme
(define-public ...)
```

Equivalent to `(begin (define foo ...) (export foo))`.

```scheme
(re-export variable ...)
```

Add all variables (which must be symbols or pairs of symbols) to the list of re-exported bindings of the current module. Pairs of symbols are handled as in `export`. Re-exported bindings must be imported by the current module from some other module.

```scheme
(export! variable ...)
```

Like `export`, but marking the exported variables as replacing. Using a module with replacing bindings will cause any existing bindings to be replaced without issuing any warnings. See the discussion of `#:replace` above.

### 6.20.4 Modules and the File System

Typical programs only use a small subset of modules installed on a Guile system. In order to keep startup time down, Guile only loads modules when a program uses them, on demand.

When a program evaluates `use-modules (ice-9 popen)`, and the module is not loaded, Guile searches for a conventionally-named file from in the load path.

In this case, loading `ice-9 popen` will eventually cause Guile to run `primitive-load-path "ice-9/popen"`. `primitive-load-path` will search for a file `ice-9/popen` in the `%load-path` (see Section 6.18.7 [Load Paths], page 389). For each directory in `%load-path`, Guile will try to find the file name, concatenated with the extensions from `%load-extensions`. By default, this will cause Guile to stat `ice-9/popen.scm`, and then `ice-9/popen`. See Section 6.18.7 [Load Paths], page 389, for more on `primitive-load-path`.

If a corresponding compiled .go file is found in the `%load-compiled-path` or in the fallback path, and is as fresh as the source file, it will be loaded instead of the source file. If no compiled file is found, Guile may try to compile the source file and cache away the resulting .go file. See Section 6.18.5 [Compilation], page 386, for more on compilation.

Once Guile finds a suitable source or compiled file is found, the file will be loaded. If, after loading the file, the module under consideration is still not defined, Guile will signal an error.
For more information on where and how to install Scheme modules, See Section 4.7 [Installing Site Packages], page 57.

6.20.5 R6RS Version References

Guile’s module system includes support for locating modules based on a declared version specifier of the same form as the one described in R6RS (see Section “Library form” in The Revised6 Report on the Algorithmic Language Scheme). By using the #:version keyword in a define-module form, a module may specify a version as a list of zero or more exact, nonnegative integers.

This version can then be used to locate the module during the module search process. Client modules and callers of the use-modules function may specify constraints on the versions of target modules by providing a version reference, which has one of the following forms:

- (sub-version-reference ...)
- (and version-reference ...)
- (or version-reference ...)
- (not version-reference)

in which sub-version-reference is in turn one of:

- (sub-version)
- (>= sub-version)
- (<= sub-version)
- (and sub-version-reference ...)
- (or sub-version-reference ...)
- (not sub-version-reference)

in which sub-version is an exact, nonnegative integer as above. A version reference matches a declared module version if each element of the version reference matches a corresponding element of the module version, according to the following rules:

- The and sub-form matches a version or version element if every element in the tail of the sub-form matches the specified version or version element.
- The or sub-form matches a version or version element if any element in the tail of the sub-form matches the specified version or version element.
- The not sub-form matches a version or version element if the tail of the sub-form does not match the version or version element.
- The >= sub-form matches a version element if the element is greater than or equal to the sub-version in the tail of the sub-form.
- The <= sub-form matches a version element if the version is less than or equal to the sub-version in the tail of the sub-form.
- A sub-version matches a version element if one is eqv? to the other.

For example, a module declared as:

```
(define-module (mylib mymodule) #:version (1 2 0))
```

would be successfully loaded by any of the following use-modules expressions:

```
(use-modules ((mylib mymodule) #:version (1 2 (>= 0))))
(use-modules ((mylib mymodule) #:version (or (1 2 0) (1 2 1))))
(use-modules ((mylib mymodule) #:version ((and (>= 1) (not 2)) 2 0)))
```
6.20.6 R6RS Libraries

In addition to the API described in the previous sections, you also have the option to create modules using the portable library form described in R6RS (see Section “Library form” in The Revised^6 Report on the Algorithmic Language Scheme), and to import libraries created in this format by other programmers. Guile's R6RS library implementation takes advantage of the flexibility built into the module system by expanding the R6RS library form into a corresponding Guile define-module form that specifies equivalent import and export requirements and includes the same body expressions. The library expression:

(library (mylib (1 2))
 (export mybinding)
 (import (otherlib (3))))

is equivalent to the module definition:

(define-module (mylib)
 #:version (1 2)
 #:use-module ((otherlib) #:version (3))
 #:export (mybinding))

Central to the mechanics of R6RS libraries is the concept of import and export levels, which control the visibility of bindings at various phases of a library’s lifecycle — macros necessary to expand forms in the library’s body need to be available at expand time; variables used in the body of a procedure exported by the library must be available at runtime. R6RS specifies the optional for sub-form of an import set specification (see below) as a mechanism by which a library author can indicate that a particular library import should take place at a particular phase with respect to the lifecycle of the importing library.

Guile's library implementation uses a technique called implicit phasing (first described by Abdulaziz Ghuloum and R. Kent Dybvig), which allows the expander and compiler to automatically determine the necessary visibility of a binding imported from another library. As such, the for sub-form described below is ignored by Guile (but may be required by Schemes in which phasing is explicit).

library name (export export-spec ...) (import import-spec ...)  [Scheme Syntax]
body ...

Defines a new library with the specified name, exports, and imports, and evaluates the specified body expressions in this library’s environment.

The library name is a non-empty list of identifiers, optionally ending with a version specification of the form described above (see Section 6.20.3 [Creating Guile Modules], page 409).

Each export-spec is the name of a variable defined or imported by the library, or must take the form (rename (internal-name external-name) ...), where the identifier internal-name names a variable defined or imported by the library and external-name is the name by which the variable is seen by importing libraries.

Each import-spec must be either an import set (see below) or must be of the form (for import-set import-level ...), where each import-level is one of:

   run
   expand
   (meta level)
where level is an integer. Note that since Guile does not require explicit phase specification, any import-sets found inside of for sub-forms will be “unwrapped” during expansion and processed as if they had been specified directly.

Import sets in turn take one of the following forms:

- **library-reference**
- (library library-reference)
- (only import-set identifier ...)
- (except import-set identifier ...)
- (prefix import-set identifier)
- (rename import-set (internal-identifier external-identifier) ...)

where library-reference is a non-empty list of identifiers ending with an optional version reference (see Section 6.20.5 [R6RS Version References], page 413), and the other sub-forms have the following semantics, defined recursively on nested import-sets:

- The library sub-form is used to specify libraries for import whose names begin with the identifier “library.”
- The only sub-form imports only the specified identifiers from the given import-set.
- The except sub-form imports all of the bindings exported by import-set except for those that appear in the specified list of identifiers.
- The prefix sub-form imports all of the bindings exported by import-set, first prefixing them with the specified identifier.
- The rename sub-form imports all of the identifiers exported by import-set. The binding for each internal-identifier among these identifiers is made visible to the importing library as the corresponding external-identifier; all other bindings are imported using the names provided by import-set.

Note that because Guile translates R6RS libraries into module definitions, an import specification may be used to declare a dependency on a native Guile module — although doing so may make your libraries less portable to other Schemes.

**import** import-spec ... [Scheme Syntax]

Import into the current environment the libraries specified by the given import specifications, where each import-spec takes the same form as in the library form described above.

### 6.20.7 Variables

Each module has its own hash table, sometimes known as an obarray, that maps the names defined in that module to their corresponding variable objects.

A variable is a box-like object that can hold any Scheme value. It is said to be undefined if its box holds a special Scheme value that denotes undefined-ness (which is different from all other Scheme values, including for example #f); otherwise the variable is defined.

On its own, a variable object is anonymous. A variable is said to be bound when it is associated with a name in some way, usually a symbol in a module obarray. When this happens, the name is said to be bound to the variable, in that module.

(That’s the theory, anyway. In practice, defined-ness and bound-ness sometimes get confused, because Lisp and Scheme implementations have often conflated — or deliberately
Variables do not have a read syntax. Most commonly they are created and bound implicitly by `define` expressions: a top-level `define` expression of the form

```
(define name value)
```

creates a variable with initial value `value` and binds it to the name `name` in the current module. But they can also be created dynamically by calling one of the constructor procedures `make-variable` and `make-undefined-variable`.

```scheme
(make-undefined-variable)  ; [Scheme Procedure]
(scm_make_undefined_variable)  ; [C Function]
  Return a variable that is initially unbound.

(make-variable init)  ; [Scheme Procedure]
(scm_make_variable init)  ; [C Function]
  Return a variable initialized to value `init`.

(variable-bound? var)  ; [Scheme Procedure]
(scm_variable_bound_p var)  ; [C Function]
  Return #t if `var` is bound to a value, or #f otherwise. Throws an error if `var` is not a variable object.

(variable-ref var)  ; [Scheme Procedure]
(scm_variable_ref var)  ; [C Function]
  Dereference `var` and return its value. `var` must be a variable object; see `make-variable` and `make-undefined-variable`.

(variable-set! var val)  ; [Scheme Procedure]
(scm_variable_set_x var, val)  ; [C Function]
  Set the value of the variable `var` to `val`. `var` must be a variable object, `val` can be any value. Return an unspecified value.

(variable-unset! var)  ; [Scheme Procedure]
(scm_variable_unset_x var)  ; [C Function]
  Unset the value of the variable `var`, leaving `var` unbound.

(variable? obj)  ; [Scheme Procedure]
(scm_variable_p obj)  ; [C Function]
  Return #t if `obj` is a variable object, else return #f.
```

### 6.20.8 Module System Reflection

The previous sections have described a declarative view of the module system. You can also work with it programmatically by accessing and modifying various parts of the Scheme objects that Guile uses to implement the module system.

At any time, there is a current module. This module is the one where a top-level `define` and similar syntax will add new bindings. You can find other module objects with `resolve-module`, for example.

These module objects can be used as the second argument to `eval`.
current-module

Return the current module object.

set-current-module module

Set the current module to module and return the previous current module.

save-module-excursion thunk

Call thunk within a dynamic-wind such that the module that is current at invocation time is restored when thunk’s dynamic extent is left (see Section 6.13.10 [Dynamic Wind], page 317).

More precisely, if thunk escapes non-locally, the current module (at the time of escape) is saved, and the original current module (at the time thunk’s dynamic extent was last entered) is restored. If thunk’s dynamic extent is re-entered, then the current module is saved, and the previously saved inner module is set current again.

resolve-module name [autoload=#t] [version=#f] [#:ensure=#t]

Find the module named name and return it. When it has not already been defined and autoload is true, try to auto-load it. When it can’t be found that way either, create an empty module if ensure is true, otherwise return #f. If version is true, ensure that the resulting module is compatible with the given version reference (see Section 6.20.5 [R6RS Version References], page 413). The name is a list of symbols.

resolve-interface name [#:select=#f] [#:hide=’] [#:prefix=#f] [#:renamer=#f] [#:version=#f]

Find the module named name as with resolve-module and return its interface. The interface of a module is also a module object, but it contains only the exported bindings.

module-uses module

Return a list of the interfaces used by module.

module-use! module interface

Add interface to the front of the use-list of module. Both arguments should be module objects, and interface should very likely be a module returned by resolve-interface.

reload-module module

Revisit the source file that corresponds to module. Raises an error if no source file is associated with the given module.

As mentioned in the previous section, modules contain a mapping between identifiers (as symbols) and storage locations (as variables). Guile defines a number of procedures to allow access to this mapping. If you are programming in C, Section 6.20.9 [Accessing Modules from C], page 418.

module-variable module name

Return the variable bound to name (a symbol) in module, or #f if name is unbound.
module-add!  module name  var
[Scheme Procedure]
Define a new binding between name (a symbol) and var (a variable) in module.

module-ref  module name
[Scheme Procedure]
Look up the value bound to name in module. Like module-variable, but also does
a variable-ref on the resulting variable, raising an error if name is unbound.

module-define!  module name  value
[Scheme Procedure]
Locally bind name to value in module. If name was already locally bound in module,
i.e., defined locally and not by an imported module, the value stored in the existing
variable will be updated. Otherwise, a new variable will be added to the module, via
module-add!.

module-set!  module name  value
[Scheme Procedure]
Update the binding of name in module to value, raising an error if name is not already
bound in module.

There are many other reflective procedures available in the default environment. If you
find yourself using one of them, please contact the Guile developers so that we can commit
to stability for that interface.

6.20.9 Accessing Modules from C
The last sections have described how modules are used in Scheme code, which is the rec-
commended way of creating and accessing modules. You can also work with modules from
C, but it is more cumbersome.

The following procedures are available.

SCM  scm_c_call_with_current_module (SCM module, SCM (*func)(void *), void *data)
[C Function]
Call func and make module the current module during the call. The argument data is
passed to func. The return value of scm_c_call_with_current_module is the return
value of func.

SCM  scm_public_variable (SCM module_name, SCM name)
SCM  scm_c_public_variable (const char *module_name, const char *name)
[C Function]
Find a the variable bound to the symbol name in the public interface of the module
named module_name.

module_name should be a list of symbols, when represented as a Scheme object, or a
space-separated string, in the const char * case. See scm_c_define_module below,
for more examples.

Signals an error if no module was found with the given name. If name is not bound
in the module, just returns #f.

SCM  scm_private_variable (SCM module_name, SCM name)
SCM  scm_c_private_variable (const char *module_name, const char *name)
[C Function]
Like scm_public_variable, but looks in the internals of the module named module_name
instead of the public interface. Logically, these procedures should only be
called on modules you write.
Like \texttt{scm_public_variable} or \texttt{scm_private_variable}, but if the \texttt{name} is not bound in the module, signals an error. Returns a variable, always.

\begin{verbatim}
static SCM eval_string_var;

/* NOTE: It is important that the call to 'my_init'
    happens-before all calls to 'my_eval_string'. */
void my_init (void)
{
    eval_string_var = scm_c_public_lookup ("ice-9 eval-string",
            "eval-string");
}

SCM my_eval_string (SCM str)
{
    return scm_call_1 (scm_variable_ref (eval_string_var), str);
}
\end{verbatim}

Like \texttt{scm_public_lookup} or \texttt{scm_private_lookup}, but additionally dereferences the variable. If the variable object is unbound, signals an error. Returns the value bound to \texttt{name} in \texttt{module_name}.

In addition, there are a number of other lookup-related procedures. We suggest that you use the \texttt{scm_public_} and \texttt{scm_private_} family of procedures instead, if possible.

Like \texttt{scm_c_lookup} and \texttt{scm_lookup}, but the specified module is used instead of the current one.
SCM scm_module_variable (SCM module, SCM name)  [C Function]
Like scm_module_lookup, but if the binding does not exist, just returns #f instead of raising an error.

To define a value, use scm_define:

SCM scm_c_define (const char *name, SCM val)  [C Function]
Bind the symbol indicated by name to a variable in the current module and set that variable to val. When name is already bound to a variable, use that. Else create a new variable.

SCM scm_define (SCM name, SCM val)  [C Function]
Like scm_c_define, but the symbol is specified directly.

SCM scm_c_module_define (SCM module, const char *name, SCM val)  [C Function]
SCM scm_module_define (SCM module, SCM name, SCM val)  [C Function]
Like scm_c_define and scm_define, but the specified module is used instead of the current one.

In some rare cases, you may need to access the variable that scm_module_define would have accessed, without changing the binding of the existing variable, if one is present. In that case, use scm_module_ensure_local_variable:

SCM scm_module_ensure_local_variable (SCM module, SCM sym)  [C Function]
Like scm_module_define, but if the sym is already locally bound in that module, the variable’s existing binding is not reset. Returns a variable.

SCM scm_module_reverse_lookup (SCM module, SCM variable)  [C Function]
Find the symbol that is bound to variable in module. When no such binding is found, return #f.

SCM scm_c_define_module (const char *name, void (*init)(void *), void *data)  [C Function]
Define a new module named name and make it current while init is called, passing it data. Return the module.

The parameter name is a string with the symbols that make up the module name, separated by spaces. For example, "foo bar" names the module '(foo bar)'.

When there already exists a module named name, it is used unchanged, otherwise, an empty module is created.

SCM scm_c_resolve_module (const char *name)  [C Function]
Find the module name name and return it. When it has not already been defined, try to auto-load it. When it can’t be found that way either, create an empty module.

The name is interpreted as for scm_c_define_module.

SCM scm_c_use_module (const char *name)  [C Function]
Add the module named name to the uses list of the current module, as with (use-modules name). The name is interpreted as for scm_c_define_module.
void scm_c_export (const char *name, ...) [C Function]

Add the bindings designated by name, ... to the public interface of the current module.
The list of names is terminated by NULL.

6.20.10 provide and require

Aubrey Jaffer, mostly to support his portable Scheme library SLIB, implemented a provide/require mechanism for many Scheme implementations. Library files in SLIB provide a feature, and when user programs require that feature, the library file is loaded in.

For example, the file random.scm in the SLIB package contains the line

(provide 'random)

so to use its procedures, a user would type

(require 'random)

and they would magically become available, but still have the same names! So this method is nice, but not as good as a full-featured module system.

When SLIB is used with Guile, provide and require can be used to access its facilities.

6.20.11 Environments

Scheme, as defined in R5RS, does not have a full module system. However it does define the concept of a top-level environment. Such an environment maps identifiers (symbols) to Scheme objects such as procedures and lists: Section 3.4 [About Closure], page 26. In other words, it implements a set of bindings.

Environments in R5RS can be passed as the second argument to eval (see Section 6.18.4 [Fly Evaluation], page 384). Three procedures are defined to return environments: scheme-report-environment, null-environment and interaction-environment (see Section 6.18.4 [Fly Evaluation], page 384).

In addition, in Guile any module can be used as an R5RS environment, i.e., passed as the second argument to eval.

Note: the following two procedures are available only when the (ice-9 r5rs) module is loaded:

(use-modules (ice-9 r5rs))

scheme-report-environment version [Scheme Procedure]

null-environment version [Scheme Procedure]

version must be the exact integer ‘5’, corresponding to revision 5 of the Scheme report (the Revised“5 Report on Scheme). scheme-report-environment returns a specifier for an environment that is empty except for all bindings defined in the report that are either required or both optional and supported by the implementation. null-environment returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in the report that are either required or both optional and supported by the implementation.

Currently Guile does not support values of version for other revisions of the report.

The effect of assigning (through the use of eval) a variable bound in a scheme-report-environment (for example car) is unspecified. Currently the environments specified by scheme-report-environment are not immutable in Guile.
6.21 Foreign Function Interface

The more one hacks in Scheme, the more one realizes that there are actually two computational worlds: one which is warm and alive, that land of parentheses, and one cold and dead, the land of C and its ilk.

But yet we as programmers live in both worlds, and Guile itself is half implemented in C. So it is that Guile's living half pays respect to its dead counterpart, via a spectrum of interfaces to C ranging from dynamic loading of Scheme primitives to dynamic binding of stock C library procedures.

6.21.1 Foreign Libraries

Most modern Unices have something called shared libraries. This ordinarily means that they have the capability to share the executable image of a library between several running programs to save memory and disk space. But generally, shared libraries give a lot of additional flexibility compared to the traditional static libraries. In fact, calling them 'dynamic' libraries is as correct as calling them 'shared'.

Shared libraries really give you a lot of flexibility in addition to the memory and disk space savings. When you link a program against a shared library, that library is not closely incorporated into the final executable. Instead, the executable of your program only contains enough information to find the needed shared libraries when the program is actually run. Only then, when the program is starting, is the final step of the linking process performed. This means that you need not recompile all programs when you install a new, only slightly modified version of a shared library. The programs will pick up the changes automatically the next time they are run.

Now, when all the necessary machinery is there to perform part of the linking at runtime, why not take the next step and allow the programmer to explicitly take advantage of it from within their program? Of course, many operating systems that support shared libraries do just that, and chances are that Guile will allow you to access this feature from within your Scheme programs. As you might have guessed already, this feature is called dynamic linking.\footnote{Some people also refer to the final linking stage at program startup as 'dynamic linking', so if you want to make yourself perfectly clear, it is probably best to use the more technical term \textit{dlopening}, as suggested by Gordon Matzigkeit in his libtool documentation.}

We titled this section “foreign libraries” because although the name “foreign” doesn’t leak into the API, the world of C really is foreign to Scheme – and that estrangement extends to components of foreign libraries as well, as we see in future sections.

dynamic-link [library] \hspace{1cm} \text{[Scheme Procedure]}
scm_dynamic_link (library) \hspace{1cm} \text{[C Function]}

Find the shared library denoted by \texttt{library} (a string) and link it into the running Guile application. When everything works out, return a Scheme object suitable for representing the linked object file. Otherwise an error is thrown. How object files are searched is system dependent.

Normally, \texttt{library} is just the name of some shared library file that will be searched for in the places where shared libraries usually reside, such as in \texttt{/usr/lib} and \texttt{/usr/local/lib}.\footnote{Some people also refer to the final linking stage at program startup as 'dynamic linking', so if you want to make yourself perfectly clear, it is probably best to use the more technical term \textit{dlopening}, as suggested by Gordon Matzigkeit in his libtool documentation.}
library should not contain an extension such as .so. The correct file name extension for the host operating system is provided automatically, according to libltdl’s rules (see Section “lt_dlopenext” in Shared Library Support for GNU).

When library is omitted, a global symbol handle is returned. This handle provides access to the symbols available to the program at run-time, including those exported by the program itself and the shared libraries already loaded.

Note that on hosts that use dynamic-link libraries (DLLs), the global symbol handle may not be able to provide access to symbols from recursively-loaded DLLs. Only exported symbols from those DLLs directly loaded by the program may be available.

\[\text{dynamic-object?\ obj} \quad \text{[Scheme Procedure]}\]
\[\text{scm\_dynamic\_object\_p\ (obj)} \quad \text{[C Function]}\]
Return #t if obj is a dynamic library handle, or #f otherwise.

\[\text{dynamic-unlink\ dobj} \quad \text{[Scheme Procedure]}\]
\[\text{scm\_dynamic\_unlink\ (dobj)} \quad \text{[C Function]}\]
Unlink the indicated object file from the application. The argument dobj must have been obtained by a call to dynamic-link. After dynamic-unlink has been called on dobj, its content is no longer accessible.

\[\text{(define libgl-obj (dynamic-link "libGL"))}\]
\[\text{libgl-obj} \Rightarrow \text{#<dynamic-object "libGL">}\]
\[\text{(dynamic-unlink libGL-obj)}\]
\[\text{libGL-obj} \Rightarrow \text{#<dynamic-object "libGL" (unlinked)>}\]
As you can see, after calling dynamic-unlink on a dynamically linked library, it is marked as `(unlinked)' and you are no longer able to use it with dynamic-call, etc. Whether the library is really removed from you program is system-dependent and will generally not happen when some other parts of your program still use it.

When dynamic linking is disabled or not supported on your system, the above functions throw errors, but they are still available.

### 6.21.2 Foreign Functions

The most natural thing to do with a dynamic library is to grovel around in it for a function pointer: a foreign function. dynamic-func exists for that purpose.

\[\text{dynamic-func\ name\ dobj} \quad \text{[Scheme Procedure]}\]
\[\text{scm\_dynamic\_func\ (name, dobj)} \quad \text{[C Function]}\]
Return a “handle” for the func name in the shared object referred to by dobj. The handle can be passed to dynamic-call to actually call the function.

Regardless whether your C compiler prepends an underscore ‘_’ to the global names in a program, you should not include this underscore in name since it will be added automatically when necessary.

Guile has static support for calling functions with no arguments, dynamic-call.
\texttt{dynamic-call func dobj} \hspace{1cm} \text{[Scheme Procedure]} \\
\texttt{scm\_dynamic\_call (func, dobj)} \hspace{1cm} \text{[C Function]}

Call the C function indicated by \textit{func} and \textit{dobj}. The function is passed no arguments and its return value is ignored. When \textit{function} is something returned by \texttt{dynamic-func}, call that function and ignore \textit{dobj}. When \textit{func} is a string, look it up in \textit{dynobj}; this is equivalent to

\begin{verbatim}
(dynamic-call (dynamic-func func dobj) #f)
\end{verbatim}

\texttt{dynamic-call} is not very powerful. It is mostly intended to be used for calling specially written initialization functions that will then add new primitives to Guile. For example, we do not expect that you will dynamically link \texttt{libX11} with \texttt{dynamic-link} and then construct a beautiful graphical user interface just by using \texttt{dynamic-call}. Instead, the usual way would be to write a special Guile-to-X11 glue library that has intimate knowledge about both Guile and X11 and does whatever is necessary to make them inter-operate smoothly. This glue library could then be dynamically linked into a vanilla Guile interpreter and activated by calling its initialization function. That function would add all the new types and primitives to the Guile interpreter that it has to offer.

(There is actually another, better option: simply to create a \texttt{libX11} wrapper in Scheme via the dynamic FFI. See Section 6.21.6 [Dynamic FFI], page 433, for more information.)

Given some set of C extensions to Guile, the next logical step is to integrate these glue libraries into the module system of Guile so that you can load new primitives into a running system just as you can load new Scheme code.

\texttt{load-extension lib init} \hspace{1cm} \text{[Scheme Procedure]} \\
\texttt{scm\_load\_extension (lib, init)} \hspace{1cm} \text{[C Function]}

Load and initialize the extension designated by \textit{LIB} and \textit{INIT}. When there is no pre-registered function for \textit{LIB}/\textit{INIT}, this is equivalent to

\begin{verbatim}
(dynamic-call INIT (dynamic-link LIB))
\end{verbatim}

When there is a pre-registered function, that function is called instead.

Normally, there is no pre-registered function. This option exists only for situations where dynamic linking is unavailable or unwanted. In that case, you would statically link your program with the desired library, and register its init function right after Guile has been initialized.

As for \texttt{dynamic-link}, \textit{lib} should not contain any suffix such as .\texttt{so} (see Section 6.21.1 [Foreign Libraries], page 422). It should also not contain any directory components. Libraries that implement Guile Extensions should be put into the normal locations for shared libraries. We recommend to use the naming convention \texttt{libguile-blum} for an extension related to a module (\texttt{bla blum}).

The normal way for an extension to be used is to write a small Scheme file that defines a module, and to load the extension into this module. When the module is auto-loaded, the extension is loaded as well. For example,

\begin{verbatim}
(define-module (bla blum))

(load-extension "libguile-bla-blum" "bla_init_blum")
\end{verbatim}
6.21.3 C Extensions

The most interesting application of dynamically linked libraries is probably to use them for providing compiled code modules to Scheme programs. As much fun as programming in Scheme is, every now and then comes the need to write some low-level C stuff to make Scheme even more fun.

Not only can you put these new primitives into their own module (see the previous section), you can even put them into a shared library that is only then linked to your running Guile image when it is actually needed.

An example will hopefully make everything clear. Suppose we want to make the Bessel functions of the C library available to Scheme in the module ‘(math bessel)’. First we need to write the appropriate glue code to convert the arguments and return values of the functions from Scheme to C and back. Additionally, we need a function that will add them to the set of Guile primitives. Because this is just an example, we will only implement this for the j0 function.

```c
#include <math.h>
#include <libguile.h>

SCM j0_wrapper (SCM x)
{
    return scm_from_double (j0 (scm_to_double (x, "j0")));
}

void init_math_bessel ()
{
    scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

We can already try to bring this into action by manually calling the low level functions for performing dynamic linking. The C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux, please refer to the libtool documentation for how to create dynamically linkable libraries portably.

```
gcc -shared -o libbessel.so -fPIC bessel.c
```

Now fire up Guile:

```
(define bessel-lib (dynamic-link "/libbessel.so"))
(d动态-call "init_math_bessel" bessel-lib)
(j0 2)
⇒ 0.223890779141236
```

The filename ./libbessel.so should be pointing to the shared library produced with the gcc command above, of course. The second line of the Guile interaction will call the init_math_bessel function which in turn will register the C function j0_wrapper with the Guile interpreter under the name j0. This function becomes immediately available and we can call it from Scheme.

Fun, isn’t it? But we are only half way there. This is what apropos has to say about j0:

```
(apropos "j0")
⇒ (guile-user): j0  #<primitive-procedure j0>
```
As you can see, \texttt{j0} is contained in the root module, where all the other Guile primitives like \texttt{display}, etc live. In general, a primitive is put into whatever module is the \textit{current module} at the time \texttt{scm\_c\_define\_gsubr} is called.

A compiled module should have a specially named \textit{module init function}. Guile knows about this special name and will call that function automatically after having linked in the shared library. For our example, we replace \texttt{init\_math\_bessel} with the following code in \texttt{bessel.c}:

```c
void
init_math_bessel (void *unused)
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
  scm_c_export ("j0", NULL);
}

void
scm_init_math_bessel_module ()
{
  scm_c_define_module ("math bessel", init_math_bessel, NULL);
}
```

The general pattern for the name of a module init function is: ‘\texttt{scm\_init\_}’, followed by the name of the module where the individual hierarchical components are concatenated with underscores, followed by ‘\texttt{\_module}’.

After \texttt{libbessel.so} has been rebuilt, we need to place the shared library into the right place.

Once the module has been correctly installed, it should be possible to use it like this:

```guile
guile> (load-extension "/libbessel.so" "scm\_init\_math\_bessel\_module")
guile> (use-modules (math bessel))
guile> (j0 2)
0.223890779141236
```

That’s it!

### 6.21.4 Modules and Extensions

The new primitives that you add to Guile with \texttt{scm\_c\_define\_gsubr} (see Section 6.9.2 \[Primitive Procedures\], page 249) or with any of the other mechanisms are placed into the module that is current when the \texttt{scm\_c\_define\_gsubr} is executed. Extensions loaded from the REPL, for example, will be placed into the \texttt{(guile\_user)} module, if the REPL module was not changed.

To define C primitives within a specific module, the simplest way is:

```scheme
(define-module (foo bar))
(load-extension "foobar-c-code" "foo\_bar\_init")
```

When loaded with \texttt{(use-modules \{foo bar\})}, the \texttt{load-extension} call looks for the \texttt{foobar-c-code.so} (etc) object file in Guile’s \texttt{extensiondir}, which is usually a subdirectory of the \texttt{libdir}. For example, if your \texttt{libdir} is \texttt{/usr/lib}, the \texttt{extensiondir} for the Guile 2.2.x series will be \texttt{/usr/lib/guile/2.2/}.

The extension path includes the major and minor version of Guile (the “effective version”), because Guile guarantees compatibility within a given effective version. This allows you to install different versions of the same extension for different versions of Guile.
Chapter 6: API Reference

If the extension is not found in the `extensiondir`, Guile will also search the standard system locations, such as `/usr/lib` or `/usr/local/lib`. It is preferable, however, to keep your extension out of the system library path, to prevent unintended interference with other dynamically-linked C libraries.

If someone installs your module to a non-standard location then the object file won’t be found. You can address this by inserting the install location in the `foo/bar.scm` file. This is convenient for the user and also guarantees the intended object is read, even if stray older or newer versions are in the loader’s path.

The usual way to specify an install location is with a prefix at the configure stage, for instance `./configure prefix=/opt` results in library files as say `/opt/lib/foobar-c-code.so`. When using Autoconf (see Section “Introduction” in *The GNU Autoconf Manual*), the library location is in a `libdir` variable. Its value is intended to be expanded by `make`, and can by substituted into a source file like `foo.scm.in`

```scheme
(define-module (foo bar))
(load-extension "XXextensiondirXX/foobar-c-code" "foo_bar_init")
```

with the following in a Makefile, using `sed` (see Section “Introduction” in SED),

```shell
foo.scm: foo.scm.in
    sed 's|XXextensiondirXX|$(libdir)/guile/2.2|' <foo.scm.in >foo.scm
```

The actual pattern `XXextensiondirXX` is arbitrary, it’s only something which doesn’t otherwise occur. If several modules need the value, it can be easier to create one `foo/config.scm` with a define of the `extensiondir` location, and use that as required.

```scheme
(define-module (foo config))
(define-public foo-config-extensiondir "XXextensiondirXX")
```

Such a file might have other locations too, for instance a data directory for auxiliary files, or `localedir` if the module has its own gettext message catalogue (see Section 6.25 [Internationalization], page 459).

It will be noted all of the above requires that the Scheme code to be found in `%load-path` (see Section 6.18.7 [Load Paths], page 389). Presently it’s left up to the system administrator or each user to augment that path when installing Guile modules in non-default locations. But having reached the Scheme code, that code should take care of hitting any of its own private files etc.

### 6.21.5 Foreign Pointers

The previous sections have shown how Guile can be extended at runtime by loading compiled C extensions. This approach is all well and good, but wouldn’t it be nice if we didn’t have to write any C at all? This section takes up the problem of accessing C values from Scheme, and the next discusses C functions.

#### 6.21.5.1 Foreign Types

The first impedance mismatch that one sees between C and Scheme is that in C, the storage locations (variables) are typed, but in Scheme types are associated with values, not variables. See Section 3.1.2 [Values and Variables], page 15.

So when describing a C function or a C structure so that it can be accessed from Scheme, the data types of the parameters or fields must be passed explicitly.
These “C type values” may be constructed using the constants and procedures from the (system foreign) module, which may be loaded like this:

(use-modules (system foreign))

(system foreign) exports a number of values expressing the basic C types:

int8                      [Scheme Variable]
uint8                      [Scheme Variable]
uint16                     [Scheme Variable]
int16                      [Scheme Variable]
uint32                     [Scheme Variable]
int32                      [Scheme Variable]
uint64                     [Scheme Variable]
int64                      [Scheme Variable]
float                      [Scheme Variable]
double                     [Scheme Variable]

These values represent the C numeric types of the specified sizes and signednesses.

In addition there are some convenience bindings for indicating types of platform-dependent size:

int                      [Scheme Variable]
unsigned-int              [Scheme Variable]
long                      [Scheme Variable]
unsigned-long             [Scheme Variable]
short                     [Scheme Variable]
unsigned-short            [Scheme Variable]
size_t                    [Scheme Variable]
ssize_t                   [Scheme Variable]
ptrdiff_t                 [Scheme Variable]
uintptr_t                 [Scheme Variable]

Values exported by the (system foreign) module, representing C numeric types.

For example, long may be equal? to int64 on a 64-bit platform.

void                      [Scheme Variable]

The void type. It can be used as the first argument to pointer->procedure to wrap a C function that returns nothing.

In addition, the symbol * is used by convention to denote pointer types. Procedures detailed in the following sections, such as pointer->procedure, accept it as a type descriptor.

6.21.5.2 Foreign Variables

Pointers to variables in the current address space may be looked up dynamically using dynamic-pointer.
dynamic-pointer name dobj

scm_dynamic_pointer (name, dobj)

Return a “wrapped pointer” for the symbol name in the shared object referred to by dobj. The returned pointer points to a C object.

Regardless whether your C compiler prepends an underscore ‘_’ to the global names in a program, you should not include this underscore in name since it will be added automatically when necessary.

For example, currently Guile has a variable, scm_numptob, as part of its API. It is declared as a C long. So, to create a handle pointing to that foreign value, we do:

```
(use-modules (system foreign))
(define numptob (dynamic-pointer "scm_numptob" (dynamic-link)))
```

numptob ⇒ #<pointer 0x7fb35b1b4688>

(The next section discusses ways to dereference pointers.)

A value returned by dynamic-pointer is a Scheme wrapper for a C pointer.

pointer-address pointer

scm_pointer_address (pointer)

Return the numerical value of pointer.

```
(pointer-address numptob)
⇒ 139984413364296 ; YMMV
```

make-pointer address [finalizer]

Return a foreign pointer object pointing to address. If finalizer is passed, it should be a pointer to a one-argument C function that will be called when the pointer object becomes unreachable.

pointer? obj

Return #t if obj is a pointer object, #f otherwise.

%null-pointer

A foreign pointer whose value is 0.

null-pointer? pointer

Return #t if pointer is the null pointer, #f otherwise.

For the purpose of passing SCM values directly to foreign functions, and allowing them to return SCM values, Guile also supports some unsafe casting operators.

scm->pointer scm

Return a foreign pointer object with the object-address of scm.

pointer->scm pointer

Unsafely cast pointer to a Scheme object. Cross your fingers!

Sometimes you want to give C extensions access to the dynamic FFI. At that point, the names get confusing, because “pointer” can refer to a SCM object that wraps a pointer, or to a void* value. We will try to use “pointer object” to refer to Scheme objects, and “pointer value” to refer to void * values.
SCM scm_from_pointer (void *ptr, void (*finalizer) (void*))  
Create a pointer object from a pointer value.
If finalizer is non-null, Guile arranges to call it on the pointer value at some point after the pointer object becomes collectable.

void* scm_to_pointer (SCM obj)  
Unpack the pointer value from a pointer object.

6.21.5.3 Void Pointers and Byte Access

Wrapped pointers are untyped, so they are essentially equivalent to C void pointers. As in C, the memory region pointed to by a pointer can be accessed at the byte level. This is achieved using bytevectors (see Section 6.6.12 [Bytevectors], page 193). The (rnrs bytevectors) module contains procedures that can be used to convert byte sequences to Scheme objects such as strings, floating point numbers, or integers.

pointer->bytevector pointer len [offset [uvec_type]]  
Return a bytevector aliasing the len bytes pointed to by pointer.

The user may specify an alternate default interpretation for the memory by passing the uvec_type argument, to indicate that the memory is an array of elements of that type. uvec_type should be something that array-type would return, like f32 or s16.
When offset is passed, it specifies the offset in bytes relative to pointer of the memory region aliased by the returned bytevector.
Mutating the returned bytevector mutates the memory pointed to by pointer, so buckle your seatbelts.

bytevector->pointer bv [offset]  
Return a pointer pointer aliasing the memory pointed to by bv or offset bytes after bv when offset is passed.

In addition to these primitives, convenience procedures are available:

dereference-pointer pointer  
Assuming pointer points to a memory region that holds a pointer, return this pointer.

string->pointer string [encoding]  
Return a foreign pointer to a nul-terminated copy of string in the given encoding, defaulting to the current locale encoding. The C string is freed when the returned foreign pointer becomes unreachable.
This is the Scheme equivalent of scm_to_stringn.

pointer->string pointer [length] [encoding]  
Return the string representing the C string pointed to by pointer. If length is omitted or -1, the string is assumed to be nul-terminated. Otherwise length is the number of bytes in memory pointed to by pointer. The C string is assumed to be in the given encoding, defaulting to the current locale encoding.
This is the Scheme equivalent of scm_from_stringn.
Most object-oriented C libraries use pointers to specific data structures to identify objects. It is useful in such cases to reify the different pointer types as disjoint Scheme types. The `define-wrapped-pointer-type` macro simplifies this.

**define-wrapped-pointer-type**

```
(define-wrapped-pointer-type type-name pred wrap unwrap       [Scheme Syntax]
    print)
```

Define helper procedures to wrap pointer objects into Scheme objects with a disjoint type. Specifically, this macro defines:

- `pred`, a predicate for the new Scheme type;
- `wrap`, a procedure that takes a pointer object and returns an object that satisfies `pred`;
- `unwrap`, which does the reverse.

`wrap` preserves pointer identity, for two pointer objects `p1` and `p2` that are `equal?`,

```
(eq? (wrap p1) (wrap p2)) ⇒ #t.
```

Finally, `print` should name a user-defined procedure to print such objects. The procedure is passed the wrapped object and a port to write to.

For example, assume we are wrapping a C library that defines a type, `bottle_t`, and functions that can be passed `bottle_t *` pointers to manipulate them. We could write:

```
(define-wrapped-pointer-type bottle
    bottle?              wrap-bottle unwrap-bottle
    (lambda (b p)
        (format p "#<bottle of " ~a ~x>
            (bottle-contents b)
            (pointer-address (unwrap-bottle b))))

(define grab-bottle
    ;; Wrapper for 'bottle_t *grab (void)'.
    (let ((grab (pointer->procedure '*
                            (dynamic-func "grab_bottle" libbottle)
                            '()))
            (lambda ()
                "Return a new bottle."
                (wrap-bottle (grab)))))

(define bottle-contents
    ;; Wrapper for 'const char *bottle_contents (bottle_t *)'.
    (let ((contents (pointer->procedure '*
                                (dynamic-func "bottle_contents" libbottle)
                                '(*))
            (lambda (b)
                "Return the contents of B."
                (pointer->string (contents (unwrap-bottle b))))))
```
(write (grab-bottle))
⇒ #<bottle of Château Haut-Brion 803d36>

In this example, grab-bottle is guaranteed to return a genuine bottle object satisfying bottle?. Likewise, bottle-contents errors out when its argument is not a genuine bottle object.

Going back to the scm_numptob example above, here is how we can read its value as a C long integer:

(use-modules (rnrs bytevectors))

(bytevector-uint-ref (pointer->bytevector numptob (sizeof long))
  0 (native-endianness)
  (sizeof long))
⇒ 8

If we wanted to corrupt Guile’s internal state, we could set scm_numptob to another value; but we shouldn’t, because that variable is not meant to be set. Indeed this point applies more widely: the C API is a dangerous place to be. Not only might setting a value crash your program, simply accessing the data pointed to by a dangling pointer or similar can prove equally disastrous.

### 6.21.5.4 Foreign Structs

Finally, one last note on foreign values before moving on to actually calling foreign functions. Sometimes you need to deal with C structs, which requires interpreting each element of the struct according to the its type, offset, and alignment. Guile has some primitives to support this.

**sizeof type**  
[Scheme Procedure]

scm_sizeof (type)  
[C Function]

Return the size of type, in bytes.

- type should be a valid C type, like int. Alternately type may be the symbol *, in which case the size of a pointer is returned. type may also be a list of types, in which case the size of a struct with ABI-conventional packing is returned.

**alignof type**  
[Scheme Procedure]

scm_alignof (type)  
[C Function]

Return the alignment of type, in bytes.

- type should be a valid C type, like int. Alternately type may be the symbol *, in which case the alignment of a pointer is returned. type may also be a list of types, in which case the alignment of a struct with ABI-conventional packing is returned.

Guile also provides some convenience methods to pack and unpack foreign pointers wrapping C structs.

**make-c-struct types vals**  
[Scheme Procedure]

Create a foreign pointer to a C struct containing vals with types types.

vals and types should be lists of the same length.
parse-c-struct foreign types

Parse a foreign pointer to a C struct, returning a list of values.

types should be a list of C types.

For example, to create and parse the equivalent of a struct { int64_t a; uint8_t b; }:

\[
\begin{align*}
&\text{(parse-c-struct (make-c-struct (list int64 uint8) }
\text{ (list 300 43)) }
\text{(list int64 uint8))}
\Rightarrow (300 43)
\end{align*}
\]

As yet, Guile only has convenience routines to support conventionally-packed structs. But given the bytevector->pointer and pointer->bytevector routines, one can create and parse tightly packed structs and unions by hand. See the code for (system foreign) for details.

### 6.21.6 Dynamic FFI

Of course, the land of C is not all nouns and no verbs: there are functions too, and Guile allows you to call them.

pointer->procedure return_type func_ptr arg_types

[Scheme Procedure]

[#:return-errno?=#:f]

scm_pointer_to_procedure (return_type, func_ptr, arg_types) [C Function]

scm_pointer_to_procedure_with_errno (return_type, func_ptr, arg_types) [C Function]

arg_types should be a list of foreign types. return_type should be a foreign type. See Section 6.21.5.1 [Foreign Types], page 427, for more information on foreign types. If return_errno? is true, or when calling scm_pointer_to_procedure_with_errno, the returned procedure will return two values, with errno as the second value.

Here is a better definition of (math bessel):

(define-module (math bessel)
  #:use-module (system foreign)
  #:export (j0))

(define libm (dynamic-link "libm"))

(define j0
  (pointer->procedure double
    (dynamic-func "j0" libm)
    (list double)))

That’s it! No C at all.
Numeric arguments and return values from foreign functions are represented as Scheme values. For example, \texttt{j0} in the above example takes a Scheme number as its argument, and returns a Scheme number.

Pointers may be passed to and returned from foreign functions as well. In that case the type of the argument or return value should be the symbol \texttt{*}, indicating a pointer. For example, the following code makes \texttt{memcpy} available to Scheme:

\begin{alltt}
(define memcpy
  (let ((this (dynamic-link)))
    (pointer->procedure '*
      (dynamic-func "memcpy" this)
      (list '* '* size_t))))
\end{alltt}

To invoke \texttt{memcpy}, one must pass it foreign pointers:

\begin{alltt}
(use-modules (rnrs bytevectors))

(define src-bits
  (u8-list->bytevector '(0 1 2 3 4 5 6 7)))
(define src
  (bytevector->pointer src-bits))
(define dest
  (bytevector->pointer (make-bytevector 16 0)))

(memcpy dest src (bytevector-length src-bits))

(bytevector->u8-list (pointer->bytevector dest 16)) ⇒ (0 1 2 3 4 5 6 7 0 0 0 0 0 0 0 0)
\end{alltt}

One may also pass structs as values, passing structs as foreign pointers. See Section 6.21.5.4 [Foreign Structs], page 432, for more information on how to express struct types and struct values.

"Out" arguments are passed as foreign pointers. The memory pointed to by the foreign pointer is mutated in place.

\begin{verbatim}
;; struct timeval {
;;    time_t tv_sec; /* seconds */
;;    suseconds_t tv_usec; /* microseconds */
;;};
;; assuming fields are of type "long"
\end{verbatim}

\begin{alltt}
(define gettimeofday
  (let ((f (pointer->procedure
    int
    (dynamic-func "gettimeofday" (dynamic-link))
    (list '* '*)))))
    (lambda ()
      (let* ((timeval (make-c-struct tv-type (list 0 0)))
        (ret (f timeval %null-pointer)))
    ))
\end{alltt}
As you can see, this interface to foreign functions is at a very low, somewhat dangerous level\textsuperscript{13}.

The FFI can also work in the opposite direction: making Scheme procedures callable from C. This makes it possible to use Scheme procedures as “callbacks” expected by a C function.

\begin{verbatim}
procedure->pointer  return-type proc arg-types  [Scheme Procedure]
scm_procedure_to_pointer (return_type, proc, arg_types) [C Function]
    Return a pointer to a C function of type return-type taking arguments of types arg-types (a list) and behaving as a proxy to procedure proc. Thus proc's arity, supported argument types, and return type should match return-type and arg-types.
\end{verbatim}

As an example, here's how the C library's \texttt{qsort} array sorting function can be made accessible to Scheme (see Section “Array Sort Function” in \textit{The GNU C Library Reference Manual}):

\begin{verbatim}
(define qsort!
  (let ((qsort (pointer->procedure void
                (dynamic-func "qsort"
                (dynamic-link))
                (list '* size_t size_t '*))))
    (lambda (bv compare)
;; Sort bytevector BV in-place according to comparison
;; procedure COMPARE.
    (let ((ptr (procedure->pointer int
            (lambda (x y)
            ;; X and Y are pointers so,
            ;; for convenience, dereference
            ;; them before calling COMPARE.
            (compare (dereference-uint8* x)
                    (dereference-uint8* y))
            (list '* '*))))
        (qsort (bytevector->pointer bv)
            (bytevector-length bv) 1 ;; we're sorting bytes
            ptr))))))

(define (dereference-uint8* ptr)
    ;; Helper function: dereference the byte pointed to by PTR.
    (let ((b (pointer->bytevector ptr 1)))
\end{verbatim}
\textsuperscript{13} A contribution to Guile in the form of a high-level FFI would be most welcome.
(bytevector-u8-ref b 0)))

(define bv
  ;; An unsorted array of bytes.
  (u8-list->bytevector '(7 1 127 3 5 4 77 2 9 0)))

;; Sort BV.
(qsort! bv (lambda (x y) (- x y)))

;; Let's see what the sorted array looks like:
(bytevector->u8-list bv)
⇒ (0 1 2 3 4 5 7 9 77 127)

And voilà!

Note that procedure->pointer is not supported (and not defined) on a few exotic architectures. Thus, user code may need to check (defined? 'procedure->pointer). Nevertheless, it is available on many architectures, including (as of libffi 3.0.9) x86, ia64, SPARC, PowerPC, ARM, and MIPS, to name a few.

6.22 Threads, Mutexes, Asyncs and Dynamic Roots

6.22.1 Threads

Guile supports POSIX threads, unless it was configured with --without-threads or the host lacks POSIX thread support. When thread support is available, the threads feature is provided (see Section 6.23.2.1 [Feature Manipulation], page 452).

The procedures below manipulate Guile threads, which are wrappers around the system’s POSIX threads. For application-level parallelism, using higher-level constructs, such as futures, is recommended (see Section 6.22.7 [Futures], page 447).

To use these facilities, load the (ice-9 threads) module.

(use-modules (ice-9 threads))

all-threads    [Scheme Procedure]
scm_all_threads ()
  Return a list of all threads.

current-thread  [Scheme Procedure]
scm_current_thread ()
  Return the thread that called this function.

call-with-new-thread thunk [handler]  [Scheme Procedure]
  Call thunk in a new thread and with a new dynamic state, returning the new thread.
  The procedure thunk is called via with-continuation-barrier.
  When handler is specified, then thunk is called from within a catch with tag #t that has handler as its handler. This catch is established inside the continuation barrier.
  Once thunk or handler returns, the return value is made the exit value of the thread and the thread is terminated.
**SCM scm_spawn_thread** (scm_t catch_body body, void *body_data, scm_t catch_handler handler, void *handler_data)  
Call body in a new thread, passing it body_data, returning the new thread. The function body is called via scm_c_with_continuation_barrier.

When handler is non-NULL, body is called via scm_internal_catch with tag SCM_BOOL_T that has handler and handler_data as the handler and its data. This catch is established inside the continuation barrier.

Once body or handler returns, the return value is made the exit value of the thread and the thread is terminated.

thread? obj  
scm_thread_p (obj)  
Return #t if obj is a thread; otherwise, return #f.

join-thread thread [timeout [timeoutval]]  
scm_join_thread (thread)  
scm_join_thread_timed (thread, timeout, timeoutval)  
Wait for thread to terminate and return its exit value. Only threads that were created with call-with-new-thread or scm_spawn_thread can be joinable; attempting to join a foreign thread will raise an error.

When timeout is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by current-time or a pair as returned by gettimeofday. When the waiting is aborted, timeoutval is returned (if it is specified; #f is returned otherwise).

thread-exited? thread  
scm_thread_exited_p (thread)  
Return #t if thread has exited, or #f otherwise.

yield  
scm_yield (thread)  
If one or more threads are waiting to execute, calling yield forces an immediate context switch to one of them. Otherwise, yield has no effect.

cancel-thread thread . values  
scm_cancel_thread (thread)  
Asynchronously interrupt thread and ask it to terminate. dynamic-wind post thunks will run, but throw handlers will not. If thread has already terminated or been signaled to terminate, this function is a no-op. Calling join-thread on the thread will return the given values, if the cancel succeeded.

Under the hood, thread cancellation uses system-async-mark and abort-to-prompt. See Section 6.22.3 [Asyncs], page 439, for more on asynchronous interrupts.

make-thread proc arg ...  
Apply proc to arg ... in a new thread formed by call-with-new-thread using a default error handler that display the error to the current error port. The arg ... expressions are evaluated in the new thread.
begin-thread expr1 expr2 . . .  

Evaluate forms expr1 expr2 . . . in a new thread formed by call-with-new-thread  
using a default error handler that display the error to the current error port.

One often wants to limit the number of threads running to be proportional to the number  
of available processors. These interfaces are therefore exported by (ice-9 threads) as well.

total-processor-count  

scm_total_processor_count ()  

Return the total number of processors of the machine, which is guaranteed to be at  
least 1. A “processor” here is a thread execution unit, which can be either:  

- an execution core in a (possibly multi-core) chip, in a (possibly multi- chip)  
  module, in a single computer, or  
- a thread execution unit inside a core in the case of hyper-threaded CPUs.

Which of the two definitions is used, is unspecified.

current-processor-count  

scm_current_processor_count ()  

Like total-processor-count, but return the number of processors available to the  
current process. See setaffinity and getaffinity for more information.

6.22.2 Thread-Local Variables

Sometimes you want to establish a variable binding that is only valid for a given thread: a  
“thread-local variable”.

You would think that fluids or parameters would be Guile’s answer for thread-local  
variables, since establishing a new fluid binding doesn’t affect bindings in other threads. See  
Section 6.13.11 [Fluids and Dynamic States], page 321, or See Section 6.13.12 [Parameters],  
page 324. However, new threads inherit the fluid bindings that were in place in their creator  
threads. In this way, a binding established using a fluid (or a parameter) in a thread can  
escape to other threads, which might not be what you want. Or, it might escape via explicit  
reification via current-dynamic-state.

Of course, this dynamic scoping might be exactly what you want; that’s why fluids and  
parameters work this way, and is what you want for for many common parameters such  
as the current input and output ports, the current locale conversion parameters, and the  
like. Perhaps this is the case for most parameters, even. If your use case for thread-local  
bindings comes from a desire to isolate a binding from its setting in unrelated threads, then  
fluids and parameters apply nicely.

On the other hand, if your use case is to prevent concurrent access to a value from  
multiple threads, then using vanilla fluids or parameters is not appropriate. For this purpose,  
Guile has thread-local fluids. A fluid created with make-thread-local-fluid won’t be  
captured by current-dynamic-state and won’t be propagated to new threads.

make-thread-local-fluid [dflt]  

scm_make_thread_local_fluid (dflt)  

Return a newly created fluid, whose initial value is dflt, or #f if dflt is not given. Unlike  
fuids made with make-fluid, thread local fluids are not captured by make-dynamic-  
state. Similarly, a newly spawned child thread does not inherit thread-local fluid  
values from the parent thread.
fluid-thread-local? fluid

scm_fluid_thread_local_p (fluid)

Return #t if the fluid fluid is is thread-local, or #f otherwise.

For example:

(define %thread-local (make-thread-local-fluid))
(with-fluids ((%thread-local (compute-data)))
  ... (fluid-ref %thread-local) ...)

You can also make a thread-local parameter out of a thread-local fluid using the normal fluid->parameter:

(define param (fluid->parameter (make-thread-local-fluid)))
(parameterize ((param (compute-data)))
  ... (param) ...)

6.22.3 Asynchronous Interrupts

Every Guile thread can be interrupted. Threads running Guile code will periodically check if there are pending interrupts and run them if necessary. To interrupt a thread, call system-async-mark on that thread.

system-async-mark proc [thread]

scm_system_async_mark (proc)

scm_system_async_mark_for_thread (proc, thread)

Enqueue proc (a procedure with zero arguments) for future execution in thread. When proc has already been enqueued for thread but has not been executed yet, this call has no effect. When thread is omitted, the thread that called system-async-mark is used.

Note that scm_system_async_mark_for_thread is not “async-signal-safe” and so cannot be called from a C signal handler. (Indeed in general, libguile functions are not safe to call from C signal handlers.)

Though an interrupt procedure can have any side effect permitted to Guile code, asynchronous interrupts are generally used either for profiling or for prematurely cancelling a computation. The former case is mostly transparent to the program being run, by design, but the latter case can introduce bugs. Like finalizers (see Section 5.5.4 [Foreign Object Memory Management], page 77), asynchronous interrupts introduce concurrency in a program. An asynchronous interrupt can run in the middle of some mutex-protected operation, for example, and potentially corrupt the program’s state.

If some bit of Guile code needs to temporarily inhibit interrupts, it can use call-with-blocked-asyns. This function works by temporarily increasing the async blocking level of the current thread while a given procedure is running. The blocking level starts out at zero, and whenever a safe point is reached, a blocking level greater than zero will prevent the execution of queued asyns.

Analogously, the procedure call-with-unblocked-asyns will temporarily decrease the blocking level of the current thread. You can use it when you want to disable asyns by default and only allow them temporarily.
In addition to the C versions of \texttt{call-with-blocked-asyns} and \texttt{call-with-unblocked-asyns}, C code can use \texttt{scm\_dynwind\_block\_asyncs} and \texttt{scm\_dynwind\_unblock\_asyncs} inside a \textit{dynamic context} (see Section 6.13.10 [Dynamic Wind], page 317) to block or unblock asyncs temporarily.

\begin{verbatim}
\begin{verbatim}
call-with-blocked-asyns proc [Scheme Procedure]
scm\_call\_with\_blocked\_asyncs (proc) [C Function]
    Call \textit{proc} and block the execution of asyncs by one level for the current thread while it is running. Return the value returned by \textit{proc}. For the first two variants, call \textit{proc} with no arguments; for the third, call it with \textit{data}.

void * scm\_c\_call\_with\_blocked\_asyncs (void *(*proc) (void *data), void *data) [C Function]
    The same but with a C function \textit{proc} instead of a Scheme thunk.

call-with-unblocked-asyns proc [Scheme Procedure]
scm\_call\_with\_unblocked\_asyncs (proc) [C Function]
    Call \textit{proc} and unblock the execution of asyncs by one level for the current thread while it is running. Return the value returned by \textit{proc}. For the first two variants, call \textit{proc} with no arguments; for the third, call it with \textit{data}.

void * scm\_c\_call\_with\_unblocked\_asyncs (void *(*proc) (void *data), void *data) [C Function]
    The same but with a C function \textit{proc} instead of a Scheme thunk.

void scm\_dynwind\_block\_asyncs () [C Function]
    During the current dynwind context, increase the blocking of asyncs by one level. This function must be used inside a pair of calls to \texttt{scm\_dynwind\_begin} and \texttt{scm\_dynwind\_end} (see Section 6.13.10 [Dynamic Wind], page 317).

void scm\_dynwind\_unblock\_asyncs () [C Function]
    During the current dynwind context, decrease the blocking of asyncs by one level. This function must be used inside a pair of calls to \texttt{scm\_dynwind\_begin} and \texttt{scm\_dynwind\_end} (see Section 6.13.10 [Dynamic Wind], page 317).
\end{verbatim}
\end{verbatim}

Sometimes you want to interrupt a thread that might be waiting for something to happen, for example on a file descriptor or a condition variable. In that case you can inform Guile of how to interrupt that wait using the following procedures:

\begin{verbatim}
\begin{verbatim}
int scm\_c\_prepare\_to\_wait\_on\_fd (int \textit{fd}) [C Function]
    Inform Guile that the current thread is about to sleep, and that if an asynchronous interrupt is signalled on this thread, Guile should wake up the thread by writing a zero byte to \textit{fd}. Returns zero if the prepare succeeded, or nonzero if the thread already has a pending async and that it should avoid waiting.

int scm\_c\_prepare\_to\_wait\_on\_cond (scm\_i\_pthread\_mutex\_t \textit{mutex}, scm\_i\_pthread\_cond\_t *\textit{cond}) [C Function]
    Inform Guile that the current thread is about to sleep, and that if an asynchronous interrupt is signalled on this thread, Guile should wake up the thread by acquiring \textit{mutex} and signalling \textit{cond}. The caller must already hold \textit{mutex} and only drop it
\end{verbatim}
\end{verbatim}
as part of the `pthread_cond_wait` call. Returns zero if the prepare succeeded, or nonzero if the thread already has a pending async and that it should avoid waiting.

```c
void scm_c_wait_finished (void) [C Function]

Inform Guile that the current thread has finished waiting, and that asynchronous interrupts no longer need any special wakeup action; the current thread will periodically poll its internal queue instead.

Guile’s own interface to `sleep`, `wait-condition-variable`, `select`, and so on all call the above routines as appropriate.

Finally, note that threads can also be interrupted via POSIX signals. See Section 7.2.8 [Signals], page 520. As an implementation detail, signal handlers will effectively call `system-async-mark` in a signal-safe way, eventually running the signal handler using the same async mechanism. In this way you can temporarily inhibit signal handlers from running using the above interfaces.

6.22.4 Atomics

When accessing data in parallel from multiple threads, updates made by one thread are not generally guaranteed to be visible by another thread. It could be that your hardware requires special instructions to be emitted to propagate a change from one CPU core to another. Or, it could be that your hardware updates values with a sequence of instructions, and a parallel thread could see a value that is in the process of being updated but not fully updated.

Atomic references solve this problem. Atomics are a standard, primitive facility to allow for concurrent access and update of mutable variables from multiple threads with guaranteed forward-progress and well-defined intermediate states.

Atomic references serve not only as a hardware memory barrier but also as a compiler barrier. Normally a compiler might choose to reorder or elide certain memory accesses due to optimizations like common subexpression elimination. Atomic accesses however will not be reordered relative to each other, and normal memory accesses will not be reordered across atomic accesses.

As an implementation detail, currently all atomic accesses and updates use the sequential consistency memory model from C11. We may relax this in the future to the acquire/release semantics, which still issues a memory barrier so that non-atomic updates are not reordered across atomic accesses or updates.

To use Guile’s atomic operations, load the `(ice-9 atomic)` module:

```scheme
(use-modules (ice-9 atomic))
```

```scheme
(make-atomic-box init) [Scheme Procedure]

Return an atomic box initialized to value `init`.

(atomic-box? obj) [Scheme Procedure]

Return `#t` if `obj` is an atomic-box object, else return `#f`.

(atomic-box-ref box) [Scheme Procedure]

Fetch the value stored in the atomic box `box` and return it.
atomic-box-set! box val
   Store val into the atomic box box.

atomic-box-swap! box val
   Store val into the atomic box box, and return the value that was previously stored in the box.

atomic-box-compare-and-swap! box expected desired
   If the value of the atomic box box is the same as, expected (in the sense of eq?), replace the contents of the box with desired. Otherwise does not update the box. Returns the previous value of the box in either case, so you can know if the swap worked by checking if the return value is eq? to expected.

6.22.5 Mutexes and Condition Variables

Mutexes are low-level primitives used to coordinate concurrent access to mutable data. Short for “mutual exclusion”, the name “mutex” indicates that only one thread at a time can acquire access to data that is protected by a mutex – threads are excluded from accessing data at the same time. If one thread has locked a mutex, then another thread attempting to lock that same mutex will wait until the first thread is done.

Mutexes can be used to build robust multi-threaded programs that take advantage of multiple cores. However, they provide very low-level functionality and are somewhat dangerous; usually you end up wanting to acquire multiple mutexes at the same time to perform a multi-object access, but this can easily lead to deadlocks if the program is not carefully written. For example, if objects A and B are protected by associated mutexes M and N, respectively, then to access both of them you need to acquire both mutexes. But what if one thread acquires M first and then N, at the same time that another thread acquires N them M? You can easily end up in a situation where one is waiting for the other.

There’s no easy way around this problem on the language level. A function A that uses mutexes does not necessarily compose nicely with a function B that uses mutexes. For this reason we suggest using atomic variables when you can (see Section 6.22.4 [Atomics], page 441), as they do not have this problem.

Still, if you as a programmer are responsible for a whole system, then you can use mutexes as a primitive to provide safe concurrent abstractions to your users. (For example, given all locks in a system, if you establish an order such that M is consistently acquired before N, you can avoid the “deadly-embrace” deadlock described above. The problem is enumerating all mutexes and establishing this order from a system perspective.) Guile gives you the low-level facilities to build such systems.

In Guile there are additional considerations beyond the usual ones in other programming languages: non-local control flow and asynchronous interrupts. What happens if you hold a mutex, but somehow you cause an exception to be thrown? There is no one right answer. You might want to keep the mutex locked to prevent any other code from ever entering that critical section again. Or, your critical section might be fine if you unlock the mutex “on the way out”, via a catch handler or dynamic-wind. See Section 6.13.8.2 [Catch], page 311, and See Section 6.13.10 [Dynamic Wind], page 317.

But if you arrange to unlock the mutex when leaving a dynamic extent via dynamic-wind, what to do if control re-enters that dynamic extent via a continuation invocation? Surely
re-entering the dynamic extent without the lock is a bad idea, so there are two options on
the table: either prevent re-entry via with-continuation-barrier or similar, or reacquire
the lock in the entry thunk of a dynamic-wind.

You might think that because you don’t use continuations, that you don’t have to think
about this, and you might be right. If you control the whole system, you can reason about
continuation use globally. Or, if you know all code that can be called in a dynamic extent,
and none of that code can call continuations, then you don’t have to worry about re-entry,
and you might not have to worry about early exit either.

However, do consider the possibility of asynchronous interrupts (see Section 6.22.3
[Asyncs], page 439). If the user interrupts your code interactively, that can cause an ex-
ception; or your thread might be cancelled, which does the same; or the user could be
running your code under some pre-emptive system that periodically causes lightweight task
switching. (Guile does not currently include such a system, but it’s possible to implement
as a library.) Probably you also want to defer asynchronous interrupt processing while you
hold the mutex, and probably that also means that you should not hold the mutex for very
long.

All of these additional Guile-specific considerations mean that from a system perspective,
you would do well to avoid these hazards if you can by not requiring mutexes. Instead, work
with immutable data that can be shared between threads without hazards, or use persistent
data structures with atomic updates based on the atomic variable library (see Section 6.22.4
[Atomics], page 441).

There are three types of mutexes in Guile: “standard”, “recursive”, and “unowned”.

Calling make-mutex with no arguments makes a standard mutex. A standard mutex can
only be locked once. If you try to lock it again from the thread that locked it to begin with
(the "owner" thread), it throws an error. It can only be unlocked from the thread that
locked it in the first place.

Calling make-mutex with the symbol recursive as the argument, or calling
make-recursive-mutex, will give you a recursive mutex. A recursive mutex can be locked
multiple times by its owner. It then has to be unlocked the corresponding number of
times, and like standard mutexes can only be unlocked by the owner thread.

Finally, calling make-mutex with the symbol allow-external-unlock creates an un-
owned mutex. An unowned mutex is like a standard mutex, except that it can be unlocked
by any thread. A corollary of this behavior is that a thread’s attempt to lock a mutex
that it already owns will block instead of signalling an error, as it could be that some other
thread unlocks the mutex, allowing the owner thread to proceed. This kind of mutex is a
bit strange and is here for use by SRFI-18.

The mutex procedures in Guile can operate on all three kinds of mutexes.

To use these facilities, load the (ice-9 threads) module.

(use-modules (ice-9 threads))

**make-mutex [kind]**  
[Scheme Procedure]

**scm_make_mutex ()**  
[C Function]

**scm_make_mutex_with_kind (SCM kind)**  
[C Function]

Return a new mutex. It will be a standard non-recursive mutex, unless the recursive
symbol is passed as the optional kind argument, in which case it will be recursive.
It’s also possible to pass `unowned` for semantics tailored to SRFI-18’s use case; see above for details.

```scheme
mutex? obj
```  
  [Scheme Procedure]

```c
scm_mutex_p (obj)
```
  [C Function]

Return `#t` if `obj` is a mutex; otherwise, return `#f`.

```scheme
make-recursive-mutex
```  
  [Scheme Procedure]

```c
scm_make_recursive_mutex ()
```
  [C Function]

Create a new recursive mutex. It is initially unlocked. Calling this function is equivalent to calling `make-mutex` with the `recursive` kind.

```scheme
lock-mutex mutex [timeout]
```  
  [Scheme Procedure]

```c
scm_lock_mutex (mutex)
```
  [C Function]

```c
scm_timed_lock_mutex (mutex, timeout)
```
  [C Function]

Lock `mutex` and return `#t`. If the mutex is already locked, then block and return only when `mutex` has been acquired.

When `timeout` is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, `#f` is returned.

For standard mutexes (`make-mutex`), an error is signalled if the thread has itself already locked `mutex`.

For a recursive mutex (`make-recursive-mutex`), if the thread has itself already locked `mutex`, then a further `lock-mutex` call increments the lock count. An additional `unlock-mutex` will be required to finally release.

When an asynchronous interrupt (see Section 6.22.3 [Asyncs], page 439) is scheduled for a thread blocked in `lock-mutex`, Guile will interrupt the wait, run the interrupts, and then resume the wait.

```c
void scm_dynwind_lock_mutex (SCM mutex)
```
  [C Function]

Arrange for `mutex` to be locked whenever the current dynwind context is entered and to be unlocked when it is exited.

```scheme
try-mutex mx
```  
  [Scheme Procedure]

```c
scm_try_mutex (mx)
```
  [C Function]

Try to lock `mutex` and return `#t` if successful, or `#f` otherwise. This is like calling `lock-mutex` with an expired timeout.

```scheme
unlock-mutex mutex
```  
  [Scheme Procedure]

```c
scm_unlock_mutex (mutex)
```
  [C Function]

Unlock `mutex`. An error is signalled if `mutex` is not locked.

“Standard” and “recursive” mutexes can only be unlocked by the thread that locked them; Guile detects this situation and signals an error. “Unowned” mutexes can be unlocked by any thread.

```scheme
mutex-owner mutex
```  
  [Scheme Procedure]

```c
scm_mutex_owner (mutex)
```
  [C Function]

Return the current owner of `mutex`, in the form of a thread or `#f` (indicating no owner). Note that a mutex may be unowned but still locked.
mutex-level \textit{mutex} \hfill [Scheme Procedure]
\texttt{scm_mutex_level (mutex)} \hfill [C Function]
Return the current lock level of \textit{mutex}. If \textit{mutex} is currently unlocked, this value will be 0; otherwise, it will be the number of times \textit{mutex} has been recursively locked by its current owner.

mutex-locked? \textit{mutex} \hfill [Scheme Procedure]
\texttt{scm_mutex_locked_p (mutex)} \hfill [C Function]
Return \#t if \textit{mutex} is locked, regardless of ownership; otherwise, return \#f.

make-condition-variable \hfill [Scheme Procedure]
\texttt{scm_make_condition_variable ()} \hfill [C Function]
Return a new condition variable.

condition-variable? \textit{obj} \hfill [Scheme Procedure]
\texttt{scm_condition_variable_p (obj)} \hfill [C Function]
Return \#t if \textit{obj} is a condition variable; otherwise, return \#f.

wait-condition-variable \textit{condvar} \textit{mutex} \textbf{[time]} \hfill [Scheme Procedure]
\texttt{scm_wait_condition_variable \textit{condvar}, \textit{mutex}, \textit{time}} \hfill [C Function]
Wait until \textit{condvar} has been signalled. While waiting, \textit{mutex} is atomically unlocked (as with unlock-mutex) and is locked again when this function returns. When \textit{time} is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by current-time or a pair as returned by gettimeofday. When the waiting is aborted, \#f is returned. When the condition variable has in fact been signalled, \#t is returned. The mutex is re-locked in any case before wait-condition-variable returns.

When an async is activated for a thread that is blocked in a call to wait-condition-variable, the waiting is interrupted, the mutex is locked, and the async is executed. When the async returns, the mutex is unlocked again and the waiting is resumed. When the thread block while re-acquiring the mutex, execution of asyncs is blocked.

signal-condition-variable \textit{condvar} \hfill [Scheme Procedure]
\texttt{scm_signal_condition_variable \textit{condvar}} \hfill [C Function]
Wake up one thread that is waiting for \textit{condvar}.

broadcast-condition-variable \textit{condvar} \hfill [Scheme Procedure]
\texttt{scm_broadcast_condition_variable \textit{condvar}} \hfill [C Function]
Wake up all threads that are waiting for \textit{condvar}.

Guile also includes some higher-level abstractions for working with mutexes.

\texttt{with-mutex \textit{mutex} body1 body2 \ldots} \hfill [macro]
Lock \textit{mutex}, evaluate the body \textit{body1 body2 \ldots}, then unlock \textit{mutex}. The return value is that returned by the last body form.

The lock, body and unlock form the branches of a \texttt{dynamic-wind} (see Section 6.13.10 [Dynamic Wind], page 317), so \textit{mutex} is automatically unlocked if an error or new continuation exits the body, and is re-locked if the body is re-entered by a captured continuation.
**monitor body1 body2 . . .** [macro]

Evaluate the body form `body1 body2 . . .` with a mutex locked so only one thread can execute that code at any one time. The return value is the return from the last body form.

Each `monitor` form has its own private mutex and the locking and evaluation is as per `with-mutex` above. A standard mutex (`make-mutex`) is used, which means the body must not recursively re-enter the `monitor` form.

The term “monitor” comes from operating system theory, where it means a particular bit of code managing access to some resource and which only ever executes on behalf of one process at any one time.

### 6.22.6 Blocking in Guile Mode

Up to Guile version 1.8, a thread blocked in guile mode would prevent the garbage collector from running. Thus threads had to explicitly leave guile mode with `scm_without_guile` () before making a potentially blocking call such as a mutex lock, a `select` () system call, etc. The following functions could be used to temporarily leave guile mode or to perform some common blocking operations in a supported way.

Starting from Guile 2.0, blocked threads no longer hinder garbage collection. Thus, the functions below are not needed anymore. They can still be used to inform the GC that a thread is about to block, giving it a (small) optimization opportunity for “stop the world” garbage collections, should they occur while the thread is blocked.

**void * scm_without_guile (void *(func) (void *), void *data)** [C Function]

Leave guile mode, call `func` on `data`, enter guile mode and return the result of calling `func`.

While a thread has left guile mode, it must not call any libguile functions except `scm_with_guile` or `scm_without_guile` and must not use any libguile macros. Also, local variables of type `SCM` that are allocated while not in guile mode are not protected from the garbage collector.

When used from non-guile mode, calling `scm_without_guile` is still allowed: it simply calls `func`. In that way, you can leave guile mode without having to know whether the current thread is in guile mode or not.

**int scm_pthread_mutex_lock (pthread_mutex_t *mutex)** [C Function]

Like `pthread_mutex_lock`, but leaves guile mode while waiting for the mutex.

**int scm_pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)** [C Function]

Like `pthread_cond_wait` and `pthread_cond_timedwait`, but leaves guile mode while waiting for the condition variable.

**int scm_pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *abstime)** [C Function]

Like `pthread_cond_wait` and `pthread_cond_timedwait`, but leaves guile mode while waiting for the condition variable.

**int scm_std_select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)** [C Function]

Like `select` but leaves guile mode while waiting. Also, the delivery of an async causes this function to be interrupted with error code `EINVAL`.
unsigned int scm_std_sleep (unsigned int seconds)  [C Function]
   Like sleep, but leaves guile mode while sleeping. Also, the delivery of an async causes
   this function to be interrupted.

unsigned long scm_std_usleep (unsigned long usecs)  [C Function]
   Like usleep, but leaves guile mode while sleeping. Also, the delivery of an async
   causes this function to be interrupted.

6.22.7 Futures
   The (ice-9 futures) module provides futures, a construct for fine-grain parallelism. A
   future is a wrapper around an expression whose computation may occur in parallel with the
code of the calling thread, and possibly in parallel with other futures. Like promises, futures
are essentially proxies that can be queried to obtain the value of the enclosed expression:

   (touch (future (+ 2 3)))
   \Rightarrow 5

   However, unlike promises, the expression associated with a future may be evaluated on
another CPU core, should one be available. This supports fine-grain parallelism, because
even relatively small computations can be embedded in futures. Consider this sequential
code:

   (define (find-prime lst1 lst2)
      (or (find prime? lst1)
           (find prime? lst2)))

   The two arms of or are potentially computation-intensive. They are independent of one
another, yet, they are evaluated sequentially when the first one returns \#f. Using futures,
one could rewrite it like this:

   (define (find-prime lst1 lst2)
      (let ((f (future (find prime? lst2))))
           (or (find prime? lst1)
                (touch f)))))

   This preserves the semantics of find-prime. On a multi-core machine, though, the
computation of (find prime? lst2) may be done in parallel with that of the other find
call, which can reduce the execution time of find-prime.

   Futures may be nested: a future can itself spawn and then touch other futures, leading
to a directed acyclic graph of futures. Using this facility, a parallel map procedure can be
defined along these lines:

   (use-modules (ice-9 futures) (ice-9 match))

   (define (par-map proc lst)
      (match lst
         (() '())
         ((head tail ...)
          (let ((tail (future (par-map proc tail)))
                 (head (proc head)))
           (cons head (touch tail)))))
Note that futures are intended for the evaluation of purely functional expressions. Expressions that have side-effects or rely on I/O may require additional care, such as explicit synchronization (see Section 6.22.5 [Mutexes and Condition Variables], page 442).

Guile’s futures are implemented on top of POSIX threads (see Section 6.22.1 [Threads], page 436). Internally, a fixed-size pool of threads is used to evaluate futures, such that offloading the evaluation of an expression to another thread doesn’t incur thread creation costs. By default, the pool contains one thread per available CPU core, minus one, to account for the main thread. The number of available CPU cores is determined using current-processor-count (see Section 7.2.7 [Processes], page 514).

When a thread touches a future that has not completed yet, it processes any pending future while waiting for it to complete, or just waits if there are no pending futures. When touch is called from within a future, the execution of the calling future is suspended, allowing its host thread to process other futures, and resumed when the touched future has completed. This suspend/resume is achieved by capturing the calling future’s continuation, and later reinstating it (see Section 6.13.5 [Prompts], page 303).

future exp
Return a future for expression exp. This is equivalent to:

(make-future (lambda () exp))

make-future thunk
Return a future for thunk, a zero-argument procedure.

This procedure returns immediately. Execution of thunk may begin in parallel with the calling thread’s computations, if idle CPU cores are available, or it may start when touch is invoked on the returned future.

If the execution of thunk throws an exception, that exception will be re-thrown when touch is invoked on the returned future.

future? obj
Return #t if obj is a future.

touch f
Return the result of the expression embedded in future f.

If the result was already computed in parallel, touch returns instantaneously. Otherwise, it waits for the computation to complete, if it already started, or initiates it. In the former case, the calling thread may process other futures in the meantime.

6.22.8 Parallel forms

The functions described in this section are available from

(use-modules (ice-9 threads))

They provide high-level parallel constructs. The following functions are implemented in terms of futures (see Section 6.22.7 [Futures], page 447). Thus they are relatively cheap as they re-use existing threads, and portable, since they automatically use one thread per available CPU core.
parallel \( expr \ldots \)  
Evaluate each \( expr \) expression in parallel, each in its own thread. Return the results of \( n \) expressions as a set of \( n \) multiple values (see Section 6.13.7 [Multiple Values], page 309).

\texttt{letpar (\((\text{var} expr) \ldots\)) body1 body2 \ldots}\)  
Evaluate each \( expr \) in parallel, each in its own thread, then bind the results to the corresponding \( \text{var} \) variables, and then evaluate \( body1 body2 \ldots \)

\texttt{letpar} is like \texttt{let} (see Section 6.12.2 [Local Bindings], page 295), but all the expressions for the bindings are evaluated in parallel.

\texttt{par-map proc lst1 lst2 \ldots}\)  
\texttt{par-for-each proc lst1 lst2 \ldots}\)  
\texttt{par-map} returns a list comprising the return values from \texttt{proc}. \texttt{par-for-each} returns an unspecified value, but waits for all calls to complete.

The \texttt{proc} calls are \((\text{proc} \text{elem1 elem2} \ldots)\), where each \texttt{elem} is from the corresponding \( \text{lst} \). Each \( \text{lst} \) must be the same length. The calls are potentially made in parallel, depending on the number of CPU cores available.

These functions are like \texttt{map} and \texttt{for-each} (see Section 6.6.9.8 [List Mapping], page 186), but make their \texttt{proc} calls in parallel.

Unlike those above, the functions described below take a number of threads as an argument. This makes them inherently non-portable since the specified number of threads may differ from the number of available CPU cores as returned by \texttt{current-processor-count} (see Section 7.2.7 [Processes], page 514). In addition, these functions create the specified number of threads when they are called and terminate them upon completion, which makes them quite expensive.

Therefore, they should be avoided.

\texttt{n-par-map n proc lst1 lst2 \ldots}\)  
\texttt{n-par-for-each n proc lst1 lst2 \ldots}\)  
\texttt{n-par-map} and \texttt{n-par-for-each} above, but use no more than \( n \) threads at any one time. The order in which calls are initiated within that threads limit is unspecified.

These functions are good for controlling resource consumption if \texttt{proc} calls might be costly, or if there are many to be made. On a dual-CPU system for instance \( n = 4 \) might be enough to keep the CPUs utilized, and not consume too much memory.

\texttt{n-for-each-par-map n sproc pproc lst1 lst2 \ldots}\)  
\texttt{n-for-each-par-map} \( n \) \texttt{sproc} \texttt{pproc} \texttt{lst1 lst2 \ldots}\)  
\texttt{n-for-each-par-map} \( n \) \texttt{sproc} \texttt{pproc} \texttt{lst1 lst2 \ldots}\)  
Apply \texttt{pproc} to the elements of the given lists, and apply \texttt{sproc} to each result returned by \texttt{pproc}. The final return value is unspecified, but all calls will have been completed before returning.

The calls made are \((\text{sproc (pproc elem1 \ldots elemN)})\), where each \texttt{elem} is from the corresponding \( \text{lst} \). Each \( \text{lst} \) must have the same number of elements.

The \texttt{pproc} calls are made in parallel, in separate threads. No more than \( n \) threads are used at any one time. The order in which \texttt{pproc} calls are initiated within that limit is unspecified.
The sproc calls are made serially, in list element order, one at a time. pproc calls on later elements may execute in parallel with the sproc calls. Exactly which thread makes each sproc call is unspecified.

This function is designed for individual calculations that can be done in parallel, but with results needing to be handled serially, for instance to write them to a file. The \( n \) limit on threads controls system resource usage when there are many calculations or when they might be costly.

It will be seen that \texttt{n-for-each-par-map} is like a combination of \texttt{n-par-map} and \texttt{for-each},
\[
  \text{(for-each sproc (n-par-map n pproc lst1 ... lstN))}
\]
But the actual implementation is more efficient since each sproc call, in turn, can be initiated once the relevant pproc call has completed, it doesn’t need to wait for all to finish.

6.23 Configuration, Features and Runtime Options

Why is my Guile different from your Guile? There are three kinds of possible variation:
- build differences — different versions of the Guile source code, installation directories, configuration flags that control pieces of functionality being included or left out, etc.
- differences in dynamically loaded code — behaviour and features provided by modules that can be dynamically loaded into a running Guile
- different runtime options — some of the options that are provided for controlling Guile’s behaviour may be set differently.

Guile provides “introspective” variables and procedures to query all of these possible variations at runtime. For runtime options, it also provides procedures to change the settings of options and to obtain documentation on what the options mean.

6.23.1 Configuration, Build and Installation

The following procedures and variables provide information about how Guile was configured, built and installed on your system.

\begin{verbatim}
version [Scheme Procedure]
effective-version [Scheme Procedure]
major-version [Scheme Procedure]
minor-version [Scheme Procedure]
micro-version [Scheme Procedure]
scm_version () [C Function]
scm_effective_version () [C Function]
scm_major_version () [C Function]
scm_minor_version () [C Function]
scm_micro_version () [C Function]
\end{verbatim}

Return a string describing Guile’s full version number, effective version number, major, minor or micro version number, respectively. The \texttt{effective-version} function returns the version name that should remain unchanged during a stable series. Currently that means that it omits the micro version. The effective version should be used for items like the versioned share directory name i.e. /usr/share/guile/2.2/
(version) ⇒ "2.2.0"
(effective-version) ⇒ "2.2"
(major-version) ⇒ "2"
(minor-version) ⇒ "2"
(micro-version) ⇒ "0"

%package-data-dir

[scheme procedure]
scm_sys_package_data_dir ()

return the name of the directory under which Guile Scheme files in general
are stored. On Unix-like systems, this is usually /usr/local/share/guile or
/usr/share/guile.

%library-dir

[scheme procedure]
scm_sys_library_dir ()

return the name of the directory where the Guile Scheme files that belong to the
core Guile installation (as opposed to files from a 3rd party package) are installed.
On Unix-like systems this is usually /usr/local/share/guile/GUILE_EFFECTIVE_
VERSION or /usr/share/guile/GUILE_EFFECTIVE_VERSION; for example /usr/local/share/guile/2.2.

%site-dir

[scheme procedure]
scm_sys_site_dir ()

return the name of the directory where Guile Scheme files specific to your site should
be installed. On Unix-like systems, this is usually /usr/local/share/guile/site
or /usr/share/guile/site.

%site-ccache-dir

[scheme procedure]
scm_sys_site_ccache_dir ()

return the directory where users should install compiled .go files for use with this
version of Guile. Might look something like /usr/lib/guile/2.2/site-ccache.

%guile-build-info

[variable]
A list of information collected during the building of a particular Guile. Entries can
be grouped into one of several categories: directories, env vars, and versioning info.
Briefly, here are the keys in %guile-build-info, by group:
directories srcdir, topsrcdir, prefix, exec_prefix, bindir, sbindir, libexecdir,
datadir, sysconfdir, sharedstatedir, localstatedir, lbindir, infodir, mandir,
includedir, pkgdatadir, pkglibdir, pkgincludedir

env vars LIBS

versioning info
guileversion, libguileinterface, buildstamp

Values are all strings. The value for LIBS is typically found also as a part of
pkg-config --libs guile-2.2 output. The value for guileversion has form X.Y.Z,
and should be the same as returned by (version). The value for libguileinterface
is libtool compatible and has form CURRENT:REVISION:AGE (see Section “Library
interface versions” in GNU Libtool). The value for buildstamp is the output of the
command ‘date -u +’%Y-%m-%d %T’’ (UTC).
In the source, `%guile-build-info` is initialized from `libguile/libpath.h`, which is completely generated, so deleting this file before a build guarantees up-to-date values for that build.

`%host-type`  
[Variable]
The canonical host type (GNU triplet) of the host Guile was configured for, e.g., "x86_64-unknown-linux-gnu" (see Section “Canonicalizing” in *The GNU Autoconf Manual*).

### 6.23.2 Feature Tracking

Guile has a Scheme level variable `*features*` that keeps track to some extent of the features that are available in a running Guile. `*features*` is a list of symbols, for example `threads`, each of which describes a feature of the running Guile process.

`*features*`  
[Variable]
A list of symbols describing available features of the Guile process.

You shouldn’t modify the `*features*` variable directly using `set!`. Instead, see the procedures that are provided for this purpose in the following subsection.

#### 6.23.2.1 Feature Manipulation

To check whether a particular feature is available, use the `provided?` procedure:

**Scheme Procedure**

```scheme
provided? feature
```

Return `#t` if the specified `feature` is available, otherwise `#f`.

To advertise a feature from your own Scheme code, you can use the `provide` procedure:

**Scheme Procedure**

```scheme
provide feature
```

Add `feature` to the list of available features in this Guile process.

For C code, the equivalent function takes its feature name as a `char *` argument for convenience:

**C Function**

```c
void scm_add_feature (const char *str)
```

Add a symbol with name `str` to the list of available features in this Guile process.

#### 6.23.2.2 Common Feature Symbols

In general, a particular feature may be available for one of two reasons. Either because the Guile library was configured and compiled with that feature enabled — i.e. the feature is built into the library on your system. Or because some C or Scheme code that was dynamically loaded by Guile has added that feature to the list.

In the first category, here are the features that the current version of Guile may define (depending on how it is built), and what they mean.

- **array** Indicates support for arrays (see Section 6.6.13 [Arrays], page 200).
- **array-for-each** Indicates availability of `array-for-each` and other array mapping procedures (see Section 6.6.13 [Arrays], page 200).
char-ready?
Indicates that the char-ready? function is available (see Section 6.14.11 [Venerable Port Interfaces], page 347).

complex 
Indicates support for complex numbers.

current-time
Indicates availability of time-related functions: times, get-internal-runtime and so on (see Section 7.2.5 [Time], page 509).

debug-extensions
Indicates that the debugging evaluator is available, together with the options for controlling it.

delay
Indicates support for promises (see Section 6.18.9 [Delayed Evaluation], page 393).

EIDs 
Indicates that the geteuid and getegid really return effective user and group IDs (see Section 7.2.7 [Processes], page 514).

inexact
Indicates support for inexact numbers.

i/o-extensions
Indicates availability of the following extended I/O procedures: ftell, redirect-port, dup->fdes, dup2, fileno, isatty?, fdopen, primitive-move->fdes and fdes->ports (see Section 7.2.2 [Ports and File Descriptors], page 493).

net-db
Indicates availability of network database functions: scm_gethost, scm_getnet, scm_getproto, scm_getserv, scm_gethost, scm_getnet, scm_setproto, scm_setserv, and their ‘byXXX’ variants (see Section 7.2.11.2 [Network Databases], page 527).

posix
Indicates support for POSIX functions: pipe, getgroups, kill, execl and so on (see Section 7.2 [POSIX], page 492).

fork
Indicates support for the POSIX fork function (see Section 7.2.7 [Processes], page 514).

popen
Indicates support for open-pipe in the (ice-9 popen) module (see Section 7.2.10 [Pipes], page 524).

random
Indicates availability of random number generation functions: random, copy-random-state, random-uniform and so on (see Section 6.6.2.14 [Random], page 127).

reckless
Indicates that Guile was built with important checks omitted — you should never see this!

regex
Indicates support for POSIX regular expressions using make-regexp, regexp-exec and friends (see Section 6.15.1 [Regexp Functions], page 356).

socket
Indicates availability of socket-related functions: socket, bind, connect and so on (see Section 7.2.11.4 [Network Sockets and Communication], page 535).

sort
Indicates availability of sorting and merging functions (see Section 6.11.3 [Sorting], page 286).
system Indicates that the system function is available (see Section 7.2.7 [Processes], page 514).

threads Indicates support for multithreading (see Section 6.22.1 [Threads], page 436).

values Indicates support for multiple return values using values and call-with-values (see Section 6.13.7 [Multiple Values], page 309).

Available features in the second category depend, by definition, on what additional code your Guile process has loaded in. The following table lists features that you might encounter for this reason.

defmacro Indicates that the defmacro macro is available (see Section 6.10 [Macros], page 261).

describe Indicates that the (oop goops describe) module has been loaded, which provides a procedure for describing the contents of GOOPS instances.

readline Indicates that Guile has loaded in Readline support, for command line editing (see Section 7.8 [Readline Support], page 695).

record Indicates support for record definition using make-record-type and friends (see Section 6.6.17 [Records], page 221).

Although these tables may seem exhaustive, it is probably unwise in practice to rely on them, as the correspondences between feature symbols and available procedures/behaviour are not strictly defined. If you are writing code that needs to check for the existence of some procedure, it is probably safer to do so directly using the defined? procedure than to test for the corresponding feature using provided?.

6.23.3 Runtime Options

There are a number of runtime options available for parameterizing built-in procedures, like read, and built-in behavior, like what happens on an uncaught error.

For more information on reader options, See Section 6.18.2 [Scheme Read], page 382.

For more information on print options, See Section 6.18.3 [Scheme Write], page 383.

Finally, for more information on debugger options, See Section 6.26.3.5 [Debug Options], page 480.

6.23.3.1 Examples of option use

Here is an example of a session in which some read and debug option handling procedures are used. In this example, the user

1. Notices that the symbols abc and aBc are not the same
2. Examines the read-options, and sees that case-insensitive is set to “no”.
3. Enables case-insensitive
4. Quits the recursive prompt
5. Verifies that now aBc and abc are the same

```
scheme@(guile-user)> (define abc "hello")
scheme@(guile-user)> abc
$1 = "hello"
scheme@(guile-user)> aBc
```
6.24 Support for Other Languages

In addition to Scheme, a user may write a Guile program in an increasing number of other languages. Currently supported languages include Emacs Lisp and ECMAScript.

Guile is still fundamentally a Scheme, but it tries to support a wide variety of language building-blocks, so that other languages can be implemented on top of Guile. This allows users to write or extend applications in languages other than Scheme, too. This section describes the languages that have been implemented.

(For details on how to implement a language, See Section 9.4 [Compiling to the Virtual Machine], page 836.)

6.24.1 Using Other Languages

There are currently only two ways to access other languages from within Guile: at the REPL, and programmatically, via compile, read-and-compile, and compile-file.

The REPL is Guile’s command prompt (see Section 4.4 [Using Guile Interactively], page 47). The REPL has a concept of the “current language”, which defaults to Scheme. The user may change that language, via the meta-command ,language.

For example, the following meta-command enables Emacs Lisp input:

```
scheme@(guile-user)> ,language elisp
```

Happy hacking with Emacs Lisp! To switch back, type ‘,L scheme’.

```
elisp@(guile-user)> (eq 1 2)
$1 = #nil
```

Each language has its short name: for example, elisp, for Elisp. The same short name may be used to compile source code programmatically, via compile:

```
elisp@(guile-user)> ,L scheme
```

Happy hacking with Guile Scheme! To switch back, type ‘,L elisp’.

```
scheme@(guile-user)> (compile '(eq 1 2) #:from 'elisp)
$2 = #nil
```
Granted, as the input to `compile` is a datum, this works best for Lispy languages, which have a straightforward datum representation. Other languages that need more parsing are better dealt with as strings.

The easiest way to deal with syntax-heavy language is with files, via `compile-file` and friends. However it is possible to invoke a language’s reader on a port, and then compile the resulting expression (which is a datum at that point). For more information, See Section 6.18.5 [Compilation], page 386.

For more details on introspecting aspects of different languages, See Section 9.4.1 [Compiler Tower], page 836.

### 6.24.2 Emacs Lisp

Emacs Lisp (Elisp) is a dynamically-scoped Lisp dialect used in the Emacs editor. See Section “Overview” in *Emacs Lisp*, for more information on Emacs Lisp.

We hope that eventually Guile’s implementation of Elisp will be good enough to replace Emacs’ own implementation of Elisp. For that reason, we have thought long and hard about how to support the various features of Elisp in a performant and compatible manner.

Readers familiar with Emacs Lisp might be curious about how exactly these various Elisp features are supported in Guile. The rest of this section focuses on addressing these concerns of the Elisp elect.

#### 6.24.2.1 Nil

nil in ELisp is an amalgam of Scheme’s `#f` and `'()`. It is false, and it is the end-of-list; thus it is a boolean, and a list as well.

Guile has chosen to support `nil` as a separate value, distinct from `#f` and `'()`. This allows existing Scheme and Elisp code to maintain their current semantics. `nil`, which in Elisp would just be written and read as `nil`, in Scheme has the external representation `#nil`.

In Elisp code, `#nil`, `#f`, and `'()` behave like `nil`, in the sense that they are all interpreted as `nil` by Elisp `if`, `cond`, `when`, `not`, `null`, etc. To test whether Elisp would interpret an object as `nil` from within Scheme code, use `nil?`:

```scheme
(nil? #nil) ⇒ #t
(nil? #f) ⇒ #t
(nil? '()) ⇒ #t
(nil? 3) ⇒ #f
```

This decision to have `nil` as a low-level distinct value facilitates interoperability between the two languages. Guile has chosen to have Scheme deal with `nil` as follows:

```scheme
(boolean? #nil) ⇒ #t
(not #nil) ⇒ #t
(null? #nil) ⇒ #t
```

And in C, one has:

```c
scm_is_bool (SCM_ELISP_NIL) ⇒ 1
```
In this way, a version of \texttt{fold} written in Scheme can correctly fold a function written in Elisp (or in fact any other language) over a nil-terminated list, as Elisp makes. The converse holds as well; a version of \texttt{fold} written in Elisp can fold over a \texttt{'}()-terminated list, as made by Scheme.

On a low level, the bit representations for \texttt{#f}, \texttt{#t}, \texttt{nil}, and \texttt{'}() are made in such a way that they differ by only one bit, and so a test for, for example, \texttt{#f}-or-\texttt{nil} may be made very efficiently. See \texttt{libguile/boolean.h}, for more information.

\section*{Equality}

Since Scheme's \texttt{equal?} must be transitive, and \texttt{'}() is not \texttt{equal?} to \texttt{#f}, to Scheme \texttt{nil} is not \texttt{equal?} to \texttt{#f} or \texttt{'}().

\begin{verbatim}
(eq? #f '()) ⇒  #f
(eq? #nil '()) ⇒  #f
(eq? #nil #f)  ⇒  #f
(eqv? #f '())  ⇒  #f
(eqv? #nil '())  ⇒  #f
(eqv? #nil #f)  ⇒  #f
(equal? #f '())  ⇒  #f
(equal? #nil '())  ⇒  #f
(equal? #nil #f)  ⇒  #f
\end{verbatim}

However, in Elisp, \texttt{'}(), \texttt{#f}, and \texttt{nil} are all \texttt{equal} (though not \texttt{eq}).

\begin{verbatim}
(defvar f (make-scheme-false))
(defvar eol (make-scheme-null))
(eq f eol)  ⇒  nil
(eq nil eol)  ⇒  nil
(eq nil f)  ⇒  nil
(equal f eol)  ⇒  t
(equal nil eol)  ⇒  t
(equal nil f)  ⇒  t
\end{verbatim}

These choices facilitate interoperability between Elisp and Scheme code, but they are not perfect. Some code that is correct standard Scheme is not correct in the presence of a second false and null value. For example:

\begin{verbatim}
(define (truthiness x)
  (if (eq? x #f)
      #f
      #t))
\end{verbatim}

This code seems to be meant to test a value for truth, but now that there are two false values, \texttt{#f} and \texttt{nil}, it is no longer correct.

Similarly, there is the loop:

\begin{verbatim}
(define (my-length 1)
  (let lp ((l 1) (len 0))
    (if (eq? l '())
      \ldots))
\end{verbatim}
len
  (lp (cdr l) (1+ len)))))

Here, \texttt{my-length} will raise an error if \texttt{l} is a \texttt{nil}-terminated list.

Both of these examples are correct standard Scheme, but, depending on what they really want to do, they are not correct Guile Scheme. Correctly written, they would test the \textit{properties} of falsehood or nullity, not the individual members of that set. That is to say, they should use \texttt{not} or \texttt{null?} to test for falsehood or nullity, not \texttt{eq?} or \texttt{memv} or the like.

Fortunately, using \texttt{not} and \texttt{null?} is in good style, so all well-written standard Scheme programs are correct, in Guile Scheme.

Here are correct versions of the above examples:

\begin{verbatim}
(define (truthiness* x)
  (if (not x)
      #f
      #t))

;; or: (define (t* x) (not (not x)))
;; or: (define (t** x) x)

(define (my-length* l)
  (let lp ((l l) (len 0))
    (if (null? l)
      len
      (lp (cdr l) (1+ len)))))
\end{verbatim}

This problem has a mirror-image case in Elisp:

\begin{verbatim}
(defun my-falsep (x)
  (if (eq x nil)
      t
      nil))
\end{verbatim}

Guile can warn when compiling code that has equality comparisons with \texttt{#f}, \texttt{'} (), or \texttt{nil}. See Section 6.18.5 [Compilation], page 386, for details.

### 6.24.2.2 Dynamic Binding

In contrast to Scheme, which uses “lexical scoping”, Emacs Lisp scopes its variables dynamically. Guile supports dynamic scoping with its “fluids” facility. See Section 6.13.11 [Fluids and Dynamic States], page 321, for more information.

### 6.24.2.3 Other Elisp Features

Buffer-local and mode-local variables should be mentioned here, along with buckybits on characters, Emacs primitive data types, the Lisp-2-ness of Elisp, and other things. Contributions to the documentation are most welcome!

### 6.24.3 ECMAScript

ECMAScript (http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf) was not the first non-Schemey language implemented by Guile, but it was the first implemented for Guile’s bytecode compiler. The goal was to support ECMAScript...
version 3.1, a relatively small language, but the implementor was completely irresponsible and got distracted by other things before finishing the standard library, and even some bits of the syntax. So, ECMAScript does deserve a mention in the manual, but it doesn’t deserve an endorsement until its implementation is completed, perhaps by some more responsible hacker.

In the meantime, the charitable user might investigate such invocations as ,L ecmascript and cat test-suite/tests/ecmascript.test.

6.25 Support for Internationalization

Guile provides internationalization\(^{14}\) support for Scheme programs in two ways. First, procedures to manipulate text and data in a way that conforms to particular cultural conventions (i.e., in a “locale-dependent” way) are provided in the (ice-9 i18n). Second, Guile allows the use of GNU gettext to translate program message strings.

6.25.1 Internationalization with Guile

In order to make use of the functions described thereafter, the (ice-9 i18n) module must be imported in the usual way:

```
(use-modules (ice-9 i18n))
```

The (ice-9 i18n) module provides procedures to manipulate text and other data in a way that conforms to the cultural conventions chosen by the user. Each region of the world or language has its own customs to, for instance, represent real numbers, classify characters, collate text, etc. All these aspects comprise the so-called “cultural conventions” of that region or language.

Computer systems typically refer to a set of cultural conventions as a locale. For each particular aspect that comprise those cultural conventions, a locale category is defined. For instance, the way characters are classified is defined by the LC_CTYPE category, while the language in which program messages are issued to the user is defined by the LC_MESSAGES category (see Section 7.2.13 [Locales], page 542, for details).

The procedures provided by this module allow the development of programs that adapt automatically to any locale setting. As we will see later, many of these procedures can optionally take a locale object argument. This additional argument defines the locale settings that must be followed by the invoked procedure. When it is omitted, then the current locale settings of the process are followed (see Section 7.2.13 [Locales], page 542).

The following procedures allow the manipulation of such locale objects.

```
make-locale category-list locale-name [base-locale]       [Scheme Procedure]
scm_make_locale (category_list, locale_name, base_locale)  [C Function]
```

Return a reference to a data structure representing a set of locale datasets. locale-name should be a string denoting a particular locale (e.g., "aa_DJ") and category-list should be either a list of locale categories or a single category as used with setlocale (see Section 7.2.13 [Locales], page 542). Optionally, if base-locale is passed, it should be a locale object denoting settings for categories not listed in category-list.

The following invocation creates a locale object that combines the use of Swedish for messages and character classification with the default settings for the other categories.

\(^{14}\) For concision and style, programmers often like to refer to internationalization as “i18n”.

(i.e., the settings of the default C locale which usually represents conventions in use in the USA):

\[ (\text{make-locale (list LC_MESSAGES LC_CTYPE) "sv_SE"}) \]

The following example combines the use of Esperanto messages and conventions with monetary conventions from Croatia:

\[ (\text{make-locale LC_MONETARY "hr_HR"}) \]
\[ (\text{make-locale LC_ALL "eo_EO"}) \]

A system-error exception (see Section 6.13.13 [Handling Errors], page 325) is raised by \text{make-locale} when \text{locale-name} does not match any of the locales compiled on the system. Note that on non-GNU systems, this error may be raised later, when the locale object is actually used.

\[ \text{locale? } \text{obj} \quad \text{[Scheme Procedure]} \]
\[ \text{scm_locale_p (obj)} \quad \text{[C Function]} \]

Return true if \text{obj} is a locale object.

\%global-locale
\[ \text{scm_global_locale} \quad \text{[C Variable]} \]

This variable is bound to a locale object denoting the current process locale as installed using \text{setlocale()} (see Section 7.2.13 [Locales], page 542). It may be used like any other locale object, including as a third argument to \text{make-locale}, for instance.

### 6.25.2 Text Collation

The following procedures provide support for text collation, i.e., locale-dependent string and character sorting.

\[ \text{string-locale<? } s1 \ s2 \ [locale] \quad \text{[Scheme Procedure]} \]
\[ \text{scm_string_locale_lt (s1, s2, locale)} \quad \text{[C Function]} \]
\[ \text{string-locale>? } s1 \ s2 \ [locale] \quad \text{[Scheme Procedure]} \]
\[ \text{scm_string_locale_gt (s1, s2, locale)} \quad \text{[C Function]} \]
\[ \text{string-locale-ci<? } s1 \ s2 \ [locale] \quad \text{[Scheme Procedure]} \]
\[ \text{scm_string_locale_ci_lt (s1, s2, locale)} \quad \text{[C Function]} \]
\[ \text{string-locale-ci>? } s1 \ s2 \ [locale] \quad \text{[Scheme Procedure]} \]
\[ \text{scm_string_locale_ci_gt (s1, s2, locale)} \quad \text{[C Function]} \]
\[ \text{Compare strings } s1 \text{ and } s2 \text{ in a locale-dependent way. If } \text{locale} \text{ is provided, it should be locale object (as returned by } \text{make-locale}) \text{ and will be used to perform the comparison; otherwise, the current system locale is used. For the } -\text{ci} \text{ variants, the comparison is made in a case-insensitive way.} \]

\[ \text{string-locale-ci=? } s1 \ s2 \ [locale] \quad \text{[Scheme Procedure]} \]
\[ \text{scm_string_locale_ci_eq (s1, s2, locale)} \quad \text{[C Function]} \]
\[ \text{Compare strings } s1 \text{ and } s2 \text{ in a case-insensitive, and locale-dependent way. If } \text{locale} \text{ is provided, it should be a locale object (as returned by } \text{make-locale}) \text{ and will be used to perform the comparison; otherwise, the current system locale is used. For the } -\text{ci} \text{ variants, the comparison is made in a case-insensitive way.} \]

\[ \text{char-locale<? } c1 \ c2 \ [locale] \quad \text{[Scheme Procedure]} \]
\[ \text{scm_char_locale_lt (c1, c2, locale)} \quad \text{[C Function]} \]
6.25.3 Character Case Mapping

The procedures below provide support for “character case mapping”, i.e., to convert characters or strings to their upper-case or lower-case equivalent. Note that SRFI-13 provides procedures that look similar (see Section 6.6.5.9 [Alphabetic Case Mapping], page 154). However, the SRFI-13 procedures are locale-independent. Therefore, they do not take into account specificities of the customs in use in a particular language or region of the world. For instance, while most languages using the Latin alphabet map lower-case letter “i” to upper-case letter “I”, Turkish maps lower-case “ı” to “Latin capital letter I with dot above”. The following procedures allow programmers to provide idiomatic character mapping.

- **char-locale-downcase chr [locale]**
  - Return the lowercase character that corresponds to `chr` according to either `locale` or the current locale.

- **char-locale-upcase chr [locale]**
  - Return the uppercase character that corresponds to `chr` according to either `locale` or the current locale.

- **char-locale-titlecase chr [locale]**
  - Return the titlecase character that corresponds to `chr` according to either `locale` or the current locale.

- **string-locale-upcase str [locale]**
  - Return a new string that is the uppercase version of `str` according to either `locale` or the current locale.

- **string-locale-downcase str [locale]**
  - Return a new string that is the down-case version of `str` according to either `locale` or the current locale.
string-locale-titlecase str [locale]  

scm_string_locale_titlecase (str, locale)  

Return a new string that is the titlecase version of str according to either locale or the current locale.

6.25.4 Number Input and Output

The following procedures allow programs to read and write numbers written according to a particular locale. As an example, in English, “ten thousand and a half” is usually written 10,000.5 while in French it is written 10 000,5. These procedures allow such differences to be taken into account.

locale-string->integer str [base [locale]]  

scm_locale_string_to_integer (str, base, locale)  

Convert string str into an integer according to either locale (a locale object as returned by make-locale) or the current process locale. If base is specified, then it determines the base of the integer being read (e.g., 16 for an hexadecimal number, 10 for a decimal number); by default, decimal numbers are read. Return two values (see Section 6.13.7 [Multiple Values], page 309): an integer (on success) or #f, and the number of characters read from str (0 on failure).

This function is based on the C library’s strtol function (see Section “Parsing of Integers” in The GNU C Library Reference Manual).

locale-string->inexact str [locale]  

scm_locale_string_to_inexact (str, locale)  

Convert string str into an inexact number according to either locale (a locale object as returned by make-locale) or the current process locale. Return two values (see Section 6.13.7 [Multiple Values], page 309): an inexact number (on success) or #f, and the number of characters read from str (0 on failure).

This function is based on the C library’s strtod function (see Section “Parsing of Floats” in The GNU C Library Reference Manual).

number->locale-string number [fraction-digits [locale]]  

Convert number (an inexact) into a string according to the cultural conventions of either locale (a locale object) or the current locale. By default, print as many fractional digits as necessary, up to an upper bound. Optionally, fraction-digits may be bound to an integer specifying the number of fractional digits to be displayed.

monetary-amount->locale-string amount intl? [locale]  

Convert amount (an inexact denoting a monetary amount) into a string according to the cultural conventions of either locale (a locale object) or the current locale. If intl? is true, then the international monetary format for the given locale is used (see Section “Currency Symbol” in The GNU C Library Reference Manual).

6.25.5 Accessing Locale Information

It is sometimes useful to obtain very specific information about a locale such as the word it uses for days or months, its format for representing floating-point figures, etc. The (ice-9
The i18n) module provides support for this in a way that is similar to the libc functions nl_langinfo () and localeconv () (see Section “Locale Information” in The GNU C Library Reference Manual). The available functions are listed below.

**locale-encoding [locale]**  
[Scheme Procedure]  
Return the name of the encoding (a string whose interpretation is system-dependent) of either locale or the current locale.

The following functions deal with dates and times.

**locale-day day [locale]**  
[Scheme Procedure]  
locale-day-short day [locale]  
[Scheme Procedure]  
locale-month month [locale]  
[Scheme Procedure]  
locale-month-short month [locale]  
[Scheme Procedure]  

Return the word (a string) used in either locale or the current locale to name the day (or month) denoted by day (or month), an integer between 1 and 7 (or 1 and 12). The -short variants provide an abbreviation instead of a full name.

**locale-am-string [locale]**  
[Scheme Procedure]  
locale-pm-string [locale]  
[Scheme Procedure]  

Return a (potentially empty) string that is used to denote ante meridiem (or post meridiem) hours in 12-hour format.

**locale-date+time-format [locale]**  
[Scheme Procedure]  
locale-date-format [locale]  
[Scheme Procedure]  
locale-time-format [locale]  
[Scheme Procedure]  
locale-time+am/pm-format [locale]  
[Scheme Procedure]  
locale-era-date-format [locale]  
[Scheme Procedure]  
locale-era-date+time-format [locale]  
[Scheme Procedure]  
locale-era-time-format [locale]  
[Scheme Procedure]  

These procedures return format strings suitable to strftime (see Section 7.2.5 [Time], page 509) that may be used to display (part of) a date/time according to certain constraints and to the conventions of either locale or the current locale (see Section “The Elegant and Fast Way” in The GNU C Library Reference Manual).

**locale-era [locale]**  
[Scheme Procedure]  
locale-era-year [locale]  
[Scheme Procedure]  

These functions return, respectively, the era and the year of the relevant era used in locale or the current locale. Most locales do not define this value. In this case, the empty string is returned. An example of a locale that does define this value is the Japanese one.

The following procedures give information about number representation.

**locale-decimal-point [locale]**  
[Scheme Procedure]  
locale-thousands-separator [locale]  
[Scheme Procedure]  

These functions return a string denoting the representation of the decimal point or that of the thousand separator (respectively) for either locale or the current locale.
locale-digit-grouping [locale] [Scheme Procedure]
Return a (potentially circular) list of integers denoting how digits of the integer part of a number are to be grouped, starting at the decimal point and going to the left. The list contains integers indicating the size of the successive groups, from right to left. If the list is non-circular, then no grouping occurs for digits beyond the last group.

For instance, if the returned list is a circular list that contains only 3 and the thousand separator is "," (as is the case with English locales), then the number 12345678 should be printed 12,345,678.

The following procedures deal with the representation of monetary amounts. Some of them take an additional intl? argument (a boolean) that tells whether the international or local monetary conventions for the given locale are to be used.

locale-monetary-decimal-point [locale] [Scheme Procedure]
locale-monetary-thousands-separator [locale] [Scheme Procedure]
locale-monetary-grouping [locale] [Scheme Procedure]
These are the monetary counterparts of the above procedures. These procedures apply to monetary amounts.

locale-currency-symbol intl? [locale] [Scheme Procedure]
Return the currency symbol (a string) of either locale or the current locale.
The following example illustrates the difference between the local and international monetary formats:

```
(define us (make-locale LC_MONETARY "en_US"))
(locale-currency-symbol #f us)  ⇒ "-$"
(locale-currency-symbol #t us)  ⇒ "USD "
```

locale-monetary-fractional-digits intl? [locale] [Scheme Procedure]
Return the number of fractional digits to be used when printing monetary amounts according to either locale or the current locale. If the locale does not specify it, then #f is returned.

locale-currency-symbol-precedes-positive? intl? [locale] [Scheme Procedure]
locale-currency-symbol-precedes-negative? intl? [locale] [Scheme Procedure]
locale-positive-separated-by-space? intl? [locale] [Scheme Procedure]
locale-negative-separated-by-space? intl? [locale] [Scheme Procedure]
These procedures return a boolean indicating whether the currency symbol should precede a positive/negative number, and whether a whitespace should be inserted between the currency symbol and a positive/negative amount.

locale-monetary-positive-sign [locale] [Scheme Procedure]
locale-monetary-negative-sign [locale] [Scheme Procedure]
Return a string denoting the positive (respectively negative) sign that should be used when printing a monetary amount.
locale-positive-sign-position
locale-negative-sign-position

These functions return a symbol telling where a sign of a positive/negative monetary amount is to appear when printing it. The possible values are:

parenthesize
The currency symbol and quantity should be surrounded by parentheses.

sign-before
Print the sign string before the quantity and currency symbol.

sign-after
Print the sign string after the quantity and currency symbol.

sign-before-currency-symbol
Print the sign string right before the currency symbol.

sign-after-currency-symbol
Print the sign string right after the currency symbol.

unspecified
Unspecified. We recommend you print the sign after the currency symbol.

Finally, the two following procedures may be helpful when programming user interfaces:

locale-yes-regexp [locale]  [Scheme Procedure]
locale-no-regexp [locale]  [Scheme Procedure]

Return a string that can be used as a regular expression to recognize a positive (respectively, negative) response to a yes/no question. For the C locale, the default values are typically "^[yY]" and "^[nN]", respectively.

Here is an example:

```
(use-modules (ice-9 rdelim))
(format #t "Does Guile rock?~%")
(let lp ((answer (read-line)))
  (cond ((string-match (locale-yes-regexp) answer)
    (format #t "High fives!~%"))
    ((string-match (locale-no-regexp) answer)
    (format #t "How about now? Does it rock yet?~%")
    (lp (read-line)))
    (else
    (format #t "What do you mean?~%")
    (lp (read-line)))))
```

For an internationalized yes/no string output, gettext should be used (see Section 6.25.6 [Gettext Support], page 466).

Example uses of some of these functions are the implementation of the number->locale-string and monetary-amount->locale-string procedures (see Section 6.25.4 [Number Input and Output], page 462), as well as that the SRFI-19 date and time conversion to/from strings (see Section 7.5.16 [SRFI-19], page 608).
6.25.6 Gettext Support

Guile provides an interface to GNU gettext for translating message strings (see Section “Introduction” in GNU gettext utilities).

Messages are collected in domains, so different libraries and programs maintain different message catalogues. The domain parameter in the functions below is a string (it becomes part of the message catalog filename).

When gettext is not available, or if Guile was configured `--without-nls`, dummy functions doing no translation are provided. When gettext support is available in Guile, the i18n feature is provided (see Section 6.23.2 [Feature Tracking], page 452).

**gettext msg [domain [category]]**  
**scm_gettext (msg, domain, category)**  
[Scheme Procedure]  
[C Function]  
Return the translation of msg in domain. domain is optional and defaults to the domain set through textdomain below. category is optional and defaults to LC_MESSAGES (see Section 7.2.13 [Locales], page 542).

Normal usage is for msg to be a literal string. xgettext can extract those from the source to form a message catalogue ready for translators (see Section “Invoking the xgettext Program” in GNU gettext utilities).

(display (gettext "You are in a maze of twisty passages."))  
_ is a commonly used shorthand, an application can make that an alias for gettext. Or a library can make a definition that uses its specific domain (so an application can change the default without affecting the library).

(define (_ msg) (gettext msg "mylibrary"))  
(display (_ "File not found."))  
_ is also a good place to perhaps strip disambiguating extra text from the message string, as for instance in Section “How to use gettext in GUI programs” in GNU gettext utilities.

**ngettext msg msgplural n [domain [category]]**  
**scm_ngettext (msg, msgplural, n, domain, category)**  
[Scheme Procedure]  
[C Function]  
Return the translation of msg / msgplural in domain, with a plural form chosen appropriately for the number n. domain is optional and defaults to the domain set through textdomain below. category is optional and defaults to LC_MESSAGES (see Section 7.2.13 [Locales], page 542).

msg is the singular form, and msgplural the plural. When no translation is available, msg is used if n = 1, or msgplural otherwise. When translated, the message catalogue can have a different rule, and can have more than two possible forms.

As per gettext above, normal usage is for msg and msgplural to be literal strings, since xgettext can extract them from the source to build a message catalogue. For example,

(define (done n)  
 (format #t (ngettext "~a file processed"  
              "~a files processed" n)  
        n))
It’s important to use `ngettext` rather than plain `gettext` for plurals, since the rules for singular and plural forms in English are not the same in other languages. Only `ngettext` will allow translators to give correct forms (see Section “Additional functions for plural forms” in GNU gettext utilities).

```scheme
(textdomain "myprog")
⇒ "myprog"
```

```scheme
(bind-textdomain-codeset "myprog" "latin-9")
⇒ "latin-9"
```

The encoding requested can be different from the translated data file, messages will be recoded as necessary. But note that when there is no translation, `gettext` returns its `msg` unchanged, ie. without any recoding. For that reason source message strings are best as plain ASCII. Currently Guile has no understanding of multi-byte characters, and string functions won’t recognise character boundaries in multi-byte strings. An application will at least be able to pass such strings through to some output though. Perhaps this will change in the future.
6.26 Debugging Infrastructure

In order to understand Guile’s debugging facilities, you first need to understand a little about how Guile represents the Scheme control stack. With that in place we explain the low level trap calls that the virtual machine can be configured to make, and the trap and breakpoint infrastructure that builds on top of those calls.

6.26.1 Evaluation and the Scheme Stack

The idea of the Scheme stack is central to a lot of debugging. The Scheme stack is a reified representation of the pending function returns in an expression’s continuation. As Guile implements function calls using a stack, this reification takes the form of a number of nested stack frames, each of which corresponds to the application of a procedure to a set of arguments.

A Scheme stack always exists implicitly, and can be summoned into concrete existence as a first-class Scheme value by the make-stack call, so that an introspective Scheme program – such as a debugger – can present it in some way and allow the user to query its details. The first thing to understand, therefore, is how Guile’s function call convention creates the stack.

Broadly speaking, Guile represents all control flow on a stack. Calling a function involves pushing an empty frame on the stack, then evaluating the procedure and its arguments, then fixing up the new frame so that it points to the old one. Frames on the stack are thus linked together. A tail call is the same, except it reuses the existing frame instead of pushing on a new one.

In this way, the only frames that are on the stack are “active” frames, frames which need to do some work before the computation is complete. On the other hand, a function that has tail-called another function will not be on the stack, as it has no work left to do.

Therefore, when an error occurs in a running program, or the program hits a breakpoint, or in fact at any point that the programmer chooses, its state at that point can be represented by a stack of all the procedure applications that are logically in progress at that time, each of which is known as a frame. The programmer can learn more about the program’s state at that point by inspecting the stack and its frames.

6.26.1.1 Stack Capture

A Scheme program can use the make-stack primitive anywhere in its code, with first arg #t, to construct a Scheme value that describes the Scheme stack at that point.

\[ \text{(make-stack \#t)} \]

\[ \Rightarrow \]

\#<stack 25205a0>

Use start-stack to limit the stack extent captured by future make-stack calls.

make-stack obj arg . . .  

[Scheme Procedure]

scm_make_stack (obj, args)  

[C Function]

Create a new stack. If obj is #t, the current evaluation stack is used for creating the stack frames, otherwise the frames are taken from obj (which must be a continuation or a frame object).
arg . . . can be any combination of integer, procedure, address range, and prompt tag values.

These values specify various ways of cutting away uninteresting stack frames from the top and bottom of the stack that make-stack returns. They come in pairs like this:

(inner_cut_1 outer_cut_1 inner_cut_2 outer_cut_2 . . .).

Each inner_cut_i can be an integer, a procedure, an address range, or a prompt tag. An integer means to cut away exactly that number of frames. A procedure means to cut away all frames up to but excluding the frame whose procedure matches the specified one. An address range is a pair of integers indicating the low and high addresses of a procedure’s code, and is the same as cutting away to a procedure (though with less work). Anything else is interpreted as a prompt tag which cuts away all frames that are inside a prompt with the given tag.

Each outer_cut_i can likewise be an integer, a procedure, an address range, or a prompt tag. An integer means to cut away that number of frames. A procedure means to cut away frames down to but excluding the frame whose procedure matches the specified one. An address range is the same, but with the procedure’s code specified as an address range. Anything else is taken to be a prompt tag, which cuts away all frames that are outside a prompt with the given tag.

If the outer_cut_i of the last pair is missing, it is taken as 0.

start-stack id exp

Evaluate exp on a new calling stack with identity id. If exp is interrupted during evaluation, backtraces will not display frames farther back than exp’s top-level form. This macro is a way of artificially limiting backtraces and stack procedures, largely as a convenience to the user.

6.26.1.2 Stacks

stack? obj

scm_stack_p (obj)

Return #t if obj is a calling stack.

stack-id stack

scm_stack_id (stack)

Return the identifier given to stack by start-stack.

stack-length stack

scm_stack_length (stack)

Return the length of stack.

stack-ref stack index

scm_stack_ref (stack, index)

Return the index’th frame from stack.

display-backtrace stack port [first [depth [highlights]]]

scm_display_backtrace_with_highlights (stack, port, first, depth, highlights)
**scm_display_backtrace** *(stack, port, first, depth)*  
[C Function]  
Display a backtrace to the output port *port*. *stack* is the stack to take the backtrace from, *first* specifies where in the stack to start and *depth* how many frames to display. *first* and *depth* can be #f, which means that default values will be used. If *highlights* is given it should be a list; the elements of this list will be highlighted wherever they appear in the backtrace.

### 6.26.1.3 Frames

**frame?** *obj*  
[Scheme Procedure]  
**scm_frame_p** *(obj)*  
[C Function]  
Return #t if *obj* is a stack frame.

**frame-previous** *frame*  
[Scheme Procedure]  
**scm_frame_previous** *(frame)*  
Return the previous frame of *frame*, or #f if *frame* is the first frame in its stack.

**frame-procedure-name** *frame*  
[Scheme Procedure]  
**scm_frame_procedure_name** *(frame)*  
Return the name of the procedure being applied in *frame*, as a symbol, or #f if the procedure has no name.

**frame-arguments** *frame*  
[Scheme Procedure]  
**scm_frame_arguments** *(frame)*  
Return the arguments of *frame*.

**frame-address** *frame*  
[Scheme Procedure]  
**frame-instruction-pointer** *frame*  
[Scheme Procedure]  
**frame-stack-pointer** *frame*  
[Scheme Procedure]  
Accessors for the three VM registers associated with this frame: the frame pointer (fp), instruction pointer (ip), and stack pointer (sp), respectively. See Section 9.3.2 [VM Concepts], page 813, for more information.

**frame-dynamic-link** *frame*  
[Scheme Procedure]  
**frame-return-address** *frame*  
[Scheme Procedure]  
**frame-mv-return-address** *frame*  
[Scheme Procedure]  
Accessors for the three saved VM registers in a frame: the previous frame pointer, the single-value return address, and the multiple-value return address. See Section 9.3.3 [Stack Layout], page 814, for more information.

**frame-bindings** *frame*  
[Scheme Procedure]  
Return a list of binding records indicating the local variables that are live in a frame.

**frame-lookup-binding** *frame* *var*  
[Scheme Procedure]  
Fetch the bindings in *frame*, and return the first one whose name is *var*, or #f otherwise.

**binding-index** *binding*  
[Scheme Procedure]  
**binding-name** *binding*  
[Scheme Procedure]  
**binding-slot** *binding*  
[Scheme Procedure]
Chapter 6: API Reference

binding-representation binding [Scheme Procedure]
Accessors for the various fields in a binding. The implicit “callee” argument is index 0, the first argument is index 1, and so on to the end of the arguments. After that are temporary variables. Note that if a variable is dead, it might not be available.

binding-ref binding [Scheme Procedure]
binding-set! binding val [Scheme Procedure]
Accessors for the values of local variables in a frame.

display-application frame [port [indent]] [Scheme Procedure]
scm_display_application (frame, port, indent) [C Function]
Display a procedure application frame to the output port port. indent specifies the indentation of the output.

Additionally, the (system vm frame) module defines a number of higher-level introspective procedures, for example to retrieve the names of local variables, and the source location to correspond to a frame. See its source code for more details.

6.26.2 Source Properties
As Guile reads in Scheme code from file or from standard input, it remembers the file name, line number and column number where each expression begins. These pieces of information are known as the source properties of the expression. Syntax expanders and the compiler propagate these source properties to compiled procedures, so that, if an error occurs when evaluating the transformed expression, Guile’s debugger can point back to the file and location where the expression originated.

The way that source properties are stored means that Guile cannot associate source properties with individual symbols, keywords, characters, booleans, or small integers. This can be seen by typing (xxx) and xxx at the Guile prompt (where the variable xxx has not been defined):

    scheme@(guile-user)> (xxx)
<unnamed port>:4:1: In procedure module-lookup:
<unnamed port>:4:1: Unbound variable: xxx

    scheme@(guile-user)> xxx
ERROR: In procedure module-lookup:
ERROR: Unbound variable: xxx

In the latter case, no source properties were stored, so the error doesn’t have any source information.

supports-source-properties? obj [Scheme Procedure]
scm_supports_source_properties_p (obj) [C Function]
Return #t if source properties can be associated with obj, otherwise return #f.

The recording of source properties is controlled by the read option named “positions” (see Section 6.18.2 [Scheme Read], page 382). This option is switched on by default.

The following procedures can be used to access and set the source properties of read expressions.
set-source-properties! \( o b j \) \( a l i s t \)

Install the association list \( a l i s t \) as the source property list for \( o b j \).

set-source-property! \( o b j \) \( k e y \) \( d a t a m \)

Set the source property of object \( o b j \), which is specified by \( k e y \) to \( d a t a m \). Normally, the key will be a symbol.

source-properties \( o b j \)

Return the source property association list of \( o b j \).

source-property \( o b j \) \( k e y \)

Return the property specified by \( k e y \) from \( o b j \)'s source properties.

If the positions reader option is enabled, supported expressions will have values set for the filename, line and column properties.

Source properties are also associated with syntax objects. Procedural macros can get at the source location of their input using the syntax-source accessor. See Section 6.10.4 [Syntax Transformer Helpers], page 272, for more.

Guile also defines a couple of convenience macros built on syntax-source:

current-source-location

Expands to the source properties corresponding to the location of the (current-source-location) form.

current-filename

Expands to the current filename: the filename that the (current-filename) form appears in. Expands to #f if this information is unavailable.

If you're stuck with defmacros (see Section 6.10.5 [Defmacros], page 275), and want to preserve source information, the following helper function might be useful to you:

cons-source \( x o r i g \) \( x \) \( y \)

Create and return a new pair whose car and cdr are \( x \) and \( y \). Any source properties associated with \( x o r i g \) are also associated with the new pair.

6.26.3 Programmatic Error Handling

For better or for worse, all programs have bugs, and dealing with bugs is part of programming. This section deals with that class of bugs that causes an exception to be raised – from your own code, from within a library, or from Guile itself.

6.26.3.1 Catching Exceptions

A common requirement is to be able to show as much useful context as possible when a Scheme program hits an error. The most immediate information about an error is the kind of error that it is – such as “division by zero” – and any parameters that the code which
signalled the error chose explicitly to provide. This information originates with the `error`
or `throw` call (or their C code equivalents, if the error is detected by C code) that signals
the error, and is passed automatically to the handler procedure of the innermost applicable
`catch` or `with-throw-handler` expression.

Therefore, to catch errors that occur within a chunk of Scheme code, and to intercept
basic information about those errors, you need to execute that code inside the dynamic
context of a `catch` or `with-throw-handler` expression, or the equivalent in C. In Scheme,
this means you need something like this:

```scheme
(catch #t
  (lambda ()
    ;; Execute the code in which
    ;; you want to catch errors here.
    ...)
  (lambda (key . parameters)
    ;; Put the code which you want
    ;; to handle an error here.
    ...))
```

The `catch` here can also be `with-throw-handler`; see Section 6.13.8.3 [Throw Handlers],
page 314, for information on the when you might want to use `with-throw-handler` instead
of `catch`.

For example, to print out a message and return `#f` when an error occurs, you might use:

```scheme
(define (catch-all thunk)
  (catch #t
    thunk
    (lambda (key . parameters)
      (format (current-error-port)
        "Uncaught throw to 'a: "a"\n" key parameters)
      #f)))

(catch-all
  (lambda () (error "Not a vegetable: tomato")))
⊣ Uncaught throw to 'misc-error: (#f ~A (Not a vegetable: tomato) #f)
⇒ #f
```

The `#t` means that the catch is applicable to all kinds of error. If you want to restrict
your catch to just one kind of error, you can put the symbol for that kind of error instead
of `#t`. The equivalent to this in C would be something like this:

```c
SCM my_body_proc (void *body_data)
{
    /* Execute the code in which
        you want to catch errors here. */
...
}

SCM my_handler_proc (void *handler_data,
                       SCM key,
                       SCM parameters)
{
    /* Put the code which you want
```
to handle an error here. */
...

{

...
scm_c_catch (SCM_BOOL_T,
    my_body_proc, body_data,
    my_handler_proc, handler_data,
    NULL, NULL);

...

} Again, as with the Scheme version, scm_c_catch could be replaced by scm_c_with_throw_handler, and SCM_BOOL_T could instead be the symbol for a particular kind of error.

6.26.3.2 Capturing the full error stack

The other interesting information about an error is the full Scheme stack at the point where the error occurred; in other words what innermost expression was being evaluated, what was the expression that called that one, and so on. If you want to write your code so that it captures and can display this information as well, there are a couple important things to understand.

Firstly, the stack at the point of the error needs to be explicitly captured by a make-stack call (or the C equivalent scm_make_stack). The Guile library does not do this “automatically” for you, so you will need to write code with a make-stack or scm_make_stack call yourself. (We emphasise this point because some people are misled by the fact that the Guile interactive REPL code does capture and display the stack automatically. But the Guile interactive REPL is itself a Scheme program running on top of the Guile library, and which uses catch and make-stack in the way we are about to describe to capture the stack when an error occurs.)

And secondly, in order to capture the stack effectively at the point where the error occurred, the make-stack call must be made before Guile unwinds the stack back to the location of the prevailing catch expression. This means that the make-stack call must be made within the handler of a with-throw-handler expression, or the optional "pre-unwind" handler of a catch. (For the full story of how these alternatives differ from each other, see Section 6.13.8 [Exceptions], page 311. The main difference is that catch terminates the error, whereas with-throw-handler only intercepts it temporarily and then allow it to continue propagating up to the next innermost handler.)

So, here are some examples of how to do all this in Scheme and in C. For the purpose of these examples we assume that the captured stack should be stored in a variable, so that it can be displayed or arbitrarily processed later on. In Scheme:

(let ((captured-stack #f))
  (catch #t
    (lambda ()

15 In effect, it is the default program which is run when no commands or script file are specified on the Guile command line.
;; Execute the code in which
;; you want to catch errors here.
(...)
(lambda (key . parameters)
  ;; Put the code which you want
  ;; to handle an error after the
  ;; stack has been unwound here.
  (...)
(lambda (key . parameters)
  ;; Capture the stack here:
  (set! captured-stack (make-stack #t)))
...
(if captured-stack
  (begin
    ;; Display or process the captured stack.
    ...))
...)

And in C:

SCM my_body_proc (void *body_data)
{
  /* Execute the code in which
     you want to catch errors here. */
  ...
}

SCM my_handler_proc (void *handler_data,
           SCM key,
           SCM parameters)
{
  /* Put the code which you want
     to handle an error after the
     stack has been unwound here. */
  ...
}

SCM my_preunwind_proc (void *handler_data,
               SCM key,
               SCM parameters)
{
  /* Capture the stack here: */
  *(SCM *)handler_data = scm_make_stack (SCM_BOOL_T, SCM_EOL);
}
{
  SCM captured_stack = SCM_BOOL_F;
  ...

Once you have a captured stack, you can interrogate and display its details in any way that you want, using the stack-... and frame-... API described in Section 6.26.1.2 [Stacks], page 469, and Section 6.26.1.3 [Frames], page 470.

If you want to print out a backtrace in the same format that the Guile REPL does, you can use the display-backtrace procedure to do so. You can also use display-application to display an individual frame in the Guile REPL format.

### 6.26.3.3 Pre-Unwind Debugging

Instead of saving a stack away and waiting for the catch to return, you can handle errors directly, from within the pre-unwind handler.

For example, to show a backtrace when an error is thrown, you might want to use a procedure like this:

```scheme
(define (with-backtrace thunk)
  (with-throw-handler #t
    thunk
    (lambda args (backtrace)))))

(with-backtrace (lambda () (error "Not a vegetable: tomato")))
```

Since we used with-throw-handler here, we didn’t actually catch the error. See Section 6.13.8.3 [Throw Handlers], page 314, for more information. However, we did print out a context at the time of the error, using the built-in procedure, backtrace.

**backtrace**  
[Scheme Procedure]

**scm_backtrace_with_highlights**  
(highlights)  
[C Function]

**scm_backtrace**  
()  
[C Function]

Display a backtrace of the current stack to the current output port. If highlights is given it should be a list; the elements of this list will be highlighted wherever they appear in the backtrace.

The Guile REPL code (in system/repl/repl.scm and related files) uses a catch with a pre-unwind handler to capture the stack when an error occurs in an expression that was typed into the REPL, and debug that stack interactively in the context of the error.

These procedures are available for use by user programs, in the (system repl error-handling) module.

```
(use-modules (system repl error-handling))
```
call-with-error-handling thunk [#:on-error [#:post-error post-error='catch] [#:pass-keys pass-keys=(quit)] [#:report-keys report-keys=(stack-overflow)] [#:trap-handler trap-handler='debug]

Call a thunk in a context in which errors are handled.

There are five keyword arguments:

- **on-error** Specifies what to do before the stack is unwound.
  - Valid options are `debug` (the default), which will enter a debugger; `pass`, in which case nothing is done, and the exception is rethrown; or a procedure, which will be the pre-unwind handler.

- **post-error** Specifies what to do after the stack is unwound.
  - Valid options are `catch` (the default), which will silently catch errors, returning the unspecified value; `report`, which prints out a description of the error (via `display-error`), and then returns the unspecified value; or a procedure, which will be the catch handler.

- **trap-handler** Specifies a trap handler: what to do when a breakpoint is hit.
  - Valid options are `debug`, which will enter the debugger; `pass`, which does nothing; or `disabled`, which disables traps entirely. See Section 6.26.4 [Traps], page 480, for more information.

- **pass-keys** A set of keys to ignore, as a list.

- **report-keys** A set of keys to always report even if the post-error handler is `catch`, as a list.

### 6.26.3.4 Stack Overflow

Every time a Scheme program makes a call that is not in tail position, it pushes a new frame onto the stack. Returning a value from a function pops the top frame off the stack. Stack frames take up memory, and as nobody has an infinite amount of memory, deep recursion could cause Guile to run out of memory. Running out of stack memory is called **stack overflow**.

#### Stack Limits

Most languages have a terrible stack overflow story. For example, in C, if you use too much stack, your program will exhibit “undefined behavior”, which if you are lucky means that it will crash. It’s especially bad in C, as you neither know ahead of time how much stack your functions use, nor the stack limit imposed by the user’s system, and the stack limit is often quite small relative to the total memory size.

Managed languages like Python have a better error story, as they are defined to raise an exception on stack overflow – but like C, Python and most dynamic languages still have a fixed stack size limit that is usually much smaller than the heap.

Arbitrary stack limits would have an unfortunate effect on Guile programs. For example, the following implementation of the inner loop of `map` is clean and elegant:

```
(define (map f l)
```
(if (pair? l)
  (cons (f (car l))
   (map f (cdr l))
  '()))

However, if there were a stack limit, that would limit the size of lists that can be processed with this map. Eventually, you would have to rewrite it to use iteration with an accumulator:

(define (map f l)
  (let lp ((l l) (out '()))
    (if (pair? l)
      (lp (cdr l) (cons (f (car l)) out))
      (reverse out)))))

This second version is sadly not as clear, and it also allocates more heap memory (once to build the list in reverse, and then again to reverse the list). You would be tempted to use the destructive reverse! to save memory and time, but then your code would not be continuation-safe – if f returned again after the map had finished, it would see an out list that had already been reversed. The recursive map has none of these problems.

Guile has no stack limit for Scheme code. When a thread makes its first Guile call, a small stack is allocated – just one page of memory. Whenever that memory limit would be reached, Guile arranges to grow the stack by a factor of two. When garbage collection happens, Guile arranges to return the unused part of the stack to the operating system, but without causing the stack to shrink. In this way, the stack can grow to consume up to all memory available to the Guile process, and when the recursive computation eventually finishes, that stack memory is returned to the system.

**Exceptional Situations**

Of course, it’s still possible to run out of stack memory. The most common cause of this is program bugs that cause unbounded recursion, as in:

(define (faulty-map f l)
  (if (pair? l)
    (cons (f (car l)) (faulty-map f l))
    '()))

Did you spot the bug? The recursive call to faulty-map recursed on l, not (cdr l). Running this program would cause Guile to use up all memory in your system, and eventually Guile would fail to grow the stack. At that point you have a problem: Guile needs to raise an exception to unwind the stack and return memory to the system, but the user might have throw handlers in place (see Section 6.13.8.3 [Throw Handlers], page 314) that want to run before the stack is unwound, and we don’t have any stack in which to run them.

Therefore in this case, Guile throws an unwind-only exception that does not run pre-unwind handlers. Because this is such an odd case, Guile prints out a message on the console, in case the user was expecting to be able to get a backtrace from any pre-unwind handler.

**Runaway Recursion**

Still, this failure mode is not so nice. If you are running an environment in which you are interactively building a program while it is running, such as at a REPL, you might want to
impose an artificial stack limit on the part of your program that you are building to detect accidental runaway recursion. For that purpose, there is `call-with-stack-overflow-handler`, from `(system vm vm)`.

(let lp () (call-with-vm lp))

`call-with-stack-overflow-handler limit thunk handler` [Scheme Procedure]
Call `thunk` in an environment in which the stack limit has been reduced to `limit` additional words. If the limit is reached, `handler` (a thunk) will be invoked in the dynamic environment of the error. For the extent of the call to `handler`, the stack limit and handler are restored to the values that were in place when `call-with-stack-overflow-handler` was called.

Usually, `handler` should raise an exception or abort to an outer prompt. However if `handler` does return, it should return a number of additional words of stack space to allow to the inner environment.

A stack overflow handler may only ever “credit” the inner thunk with stack space that was available when the handler was instated. When Guile first starts, there is no stack limit in place, so the outer handler may allow the inner thunk an arbitrary amount of space, but any nested stack overflow handler will not be able to consume more than its limit.

Unlike the unwind-only exception that is thrown if Guile is unable to grow its stack, any exception thrown by a stack overflow handler might invoke pre-unwind handlers. Indeed, the stack overflow handler is itself a pre-unwind handler of sorts. If the code imposing the stack limit wants to protect itself against malicious pre-unwind handlers from the inner thunk, it should abort to a prompt of its own making instead of throwing an exception that might be caught by the inner thunk.

**C Stack Usage**

It is also possible for Guile to run out of space on the C stack. If you call a primitive procedure which then calls a Scheme procedure in a loop, you will consume C stack space. Guile tries to detect excessive consumption of C stack space, throwing an error when you have hit 80% of the process' available stack (as allocated by the operating system), or 160 kilowords in the absence of a strict limit.

For example, looping through `call-with-vm`, a primitive that calls a thunk, gives us the following:

(let lp () (call-with-vm lp))

`ERROR: Stack overflow`

Unfortunately, that’s all the information we get. Overrunning the C stack will throw an unwind-only exception, because it’s not safe to do very much when you are close to the C stack limit.

If you get an error like this, you can either try rewriting your code to use less stack space, or increase the maximum stack size. To increase the maximum stack size, use `debug-set!`, for example:

(debug-set! stack 200000)
The next section describes `debug-set!` more thoroughly. Of course the best thing is to have your code operate without so much resource consumption by avoiding loops through C trampolines.

### 6.26.3.5 Debug options

The behavior of the `backtrace` procedure and of the default error handler can be parameterized via the debug options.

```scheme
debug-options [setting]  [Scheme Procedure]
```

Display the current settings of the debug options. If `setting` is omitted, only a short form of the current read options is printed. Otherwise if `setting` is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `debug-options` at the prompt.

```scheme
scheme@(guile-user)>
backwards no Display backtrace in anti-chronological order.
width 79 Maximal width of backtrace.
depth 20 Maximal length of printed backtrace.
backtrace yes Show backtrace on error.
stack 1048576 Stack size limit (measured in words; 0 = no check).
show-file-name #t Show file names and line numbers in backtraces when not ‘#f’. A value of ‘base’ displays only base names, while ‘#t’ displays full names.
warn-deprecated no Warn when deprecated features are used.
```

The boolean options may be toggled with `debug-enable` and `debug-disable`. The non-boolean options must be set using `debug-set!`.

```scheme
debusing-enabled option-name  [Scheme Procedure]
debusing-disable option-name  [Scheme Procedure]
debusing-set! option-name value  [Scheme Syntax]
```

Modify the debug options. `debug-enable` should be used with boolean options and switches them on, `debug-disable` switches them off.

`debug-set!` can be used to set an option to a specific value. Due to historical oddities, it is a macro that expects an unquoted option name.

### 6.26.4 Traps

Guile’s virtual machine can be configured to call out at key points to arbitrary user-specified procedures.

In principle, these hooks allow Scheme code to implement any model it chooses for examining the evaluation stack as program execution proceeds, and for suspending execution to be resumed later.

VM hooks are very low-level, though, and so Guile also has a library of higher-level traps on top of the VM hooks. A trap is an execution condition that, when fulfilled, will fire a handler. For example, Guile defines a trap that fires when control reaches a certain source location.

Finally, Guile also defines a third level of abstractions: per-thread trap states. A trap state exists to give names to traps, and to hold on to the set of traps so that they can
be enabled, disabled, or removed. The trap state infrastructure defines the most useful abstractions for most cases. For example, Guile’s REPL uses trap state functions to set breakpoints and tracepoints.

The following subsections describe all this in detail, for both the user wanting to use traps, and the developer interested in understanding how the interface hangs together.

### 6.26.4.1 VM Hooks

Everything that runs in Guile runs on its virtual machine, a C program that defines a number of operations that Scheme programs can perform.

Note that there are multiple VM “engines” for Guile. Only some of them have support for hooks compiled in. Normally the deal is that you get hooks if you are running interactively, and otherwise they are disabled, as they do have some overhead (about 10 or 20 percent).

To ensure that you are running with hooks, pass `--debug` to Guile when running your program, or otherwise use the `call-with-vm` and `set-vm-engine!` procedures to ensure that you are running in a VM with the `debug` engine.

To digress, Guile’s VM has 6 different hooks (see Section 6.11.6 [Hooks], page 288) that can be fired at different times, which may be accessed with the following procedures.

The first argument of calls to these hooks is the frame in question. See Section 6.26.1.3 [Frames], page 470. Some hooks may call their procedures with more arguments. Since these hooks may be fired very frequently, Guile does a terrible thing: it allocates the frames on the C stack instead of the garbage-collected heap.

The upshot here is that the frames are only valid within the dynamic extent of the call to the hook. If a hook procedure keeps a reference to the frame outside the extent of the hook, bad things will happen.

The interface to hooks is provided by the `(system vm vm)` module:

```scheme
(use-modules (system vm vm))
```

All of these functions implicitly act on the VM for the current thread only.

#### vm-next-hook

[Scheme Procedure]

The hook that will be fired before an instruction is retired (and executed).

#### vm-push-continuation-hook

[Scheme Procedure]

The hook that will be fired after preparing a new frame. Fires just before applying a procedure in a non-tail context, just before the corresponding apply-hook.

#### vm-pop-continuation-hook

[Scheme Procedure]

The hook that will be fired before returning from a frame.

This hook fires with a variable number of arguments, corresponding to the values that the frame returns to its continuation.

#### vm-apply-hook

[Scheme Procedure]

The hook that will be fired before a procedure is applied. The frame’s procedure will have already been set to the new procedure.

Note that procedure application is somewhat orthogonal to continuation pushes and pops. A non-tail call to a procedure will result first in a firing of the push-continuation hook, then this application hook, whereas a tail call will run without having fired a push-continuation hook.
vm-abort-continuation-hook

The hook that will be called after aborting to a prompt. See Section 6.13.5 [Prompts], page 303.

Like the pop-continuation hook, this hook fires with a variable number of arguments, corresponding to the values that returned to the continuation.

vm-restore-continuation-hook

The hook that will be called after restoring an undelimited continuation. Unfortunately it’s not currently possible to introspect on the values that were given to the continuation.

These hooks do impose a performance penalty, if they are on. Obviously, the vm-next-hook has quite an impact, performance-wise. Therefore Guile exposes a single, heavy-handed knob to turn hooks on or off, the VM trace level. If the trace level is positive, hooks run; otherwise they don’t.

For convenience, when the VM fires a hook, it does so with the trap level temporarily set to 0. That way the hooks don’t fire while you’re handling a hook. The trace level is restored to whatever it was once the hook procedure finishes.

vm-trace-level

Retrieve the “trace level” of the VM. If positive, the trace hooks associated with vm will be run. The initial trace level is 0.

set-vm-trace-level! level

Set the “trace level” of the VM.

See Section 9.3 [A Virtual Machine for Guile], page 812, for more information on Guile’s virtual machine.

6.26.4.2 Trap Interface

The capabilities provided by hooks are great, but hooks alone rarely correspond to what users want to do.

For example, if a user wants to break when and if control reaches a certain source location, how do you do it? If you install a “next” hook, you get unacceptable overhead for the execution of the entire program. It would be possible to install an “apply” hook, then if the procedure encompasses those source locations, install a “next” hook, but already you’re talking about one concept that might be implemented by a varying number of lower-level concepts.

It’s best to be clear about things and define one abstraction for all such conditions: the trap.

Considering the myriad capabilities offered by the hooks though, there is only a minimum of functionality shared by all traps. Guile’s current take is to reduce this to the absolute minimum, and have the only standard interface of a trap be “turn yourself on” or “turn yourself off”.

This interface sounds a bit strange, but it is useful to procedurally compose higher-level traps from lower-level building blocks. For example, Guile defines a trap that calls one handler when control enters a procedure, and another when control leaves the procedure.
Given that trap, one can define a trap that adds to the next-hook only when within a given procedure. Building further, one can define a trap that fires when control reaches particular instructions within a procedure.

Or of course you can stop at any of these intermediate levels. For example, one might only be interested in calls to a given procedure. But the point is that a simple enable/disable interface is all the commonality that exists between the various kinds of traps, and furthermore that such an interface serves to allow “higher-level” traps to be composed from more primitive ones.

Specifically, a trap, in Guile, is a procedure. When a trap is created, by convention the trap is enabled; therefore, the procedure that is the trap will, when called, disable the trap, and return a procedure that will enable the trap, and so on.

Trap procedures take one optional argument: the current frame. (A trap may want to add to different sets of hooks depending on the frame that is current at enable-time.)

If this all sounds very complicated, it’s because it is. Some of it is essential, but probably most of it is not. The advantage of using this minimal interface is that composability is more lexically apparent than when, for example, using a stateful interface based on GOOPS. But perhaps this reflects the cognitive limitations of the programmer who made the current interface more than anything else.

6.26.4.3 Low-Level Traps

To summarize the last sections, traps are enabled or disabled, and when they are enabled, they add to various VM hooks.

Note, however, that traps do not increase the VM trace level. So if you create a trap, it will be enabled, but unless something else increases the VM’s trace level (see Section 6.26.4.1 [VM Hooks], page 481), the trap will not fire. It turns out that getting the VM trace level right is tricky without a global view of what traps are enabled. See Section 6.26.4.5 [Trap States], page 487, for Guile’s answer to this problem.

Traps are created by calling procedures. Most of these procedures share a set of common keyword arguments, so rather than document them separately, we discuss them all together here:

#:vm The VM to instrument. Defaults to the current thread’s VM.

#:current-frame For traps that enable more hooks depending on their dynamic context, this argument gives the current frame that the trap is running in. Defaults to #f.

To have access to these procedures, you’ll need to have imported the (system vm traps) module:

(use-modules (system vm traps))

trap-at-procedure-call proc handler [#:vm] [Scheme Procedure]
A trap that calls handler when proc is applied.

trap-in-procedure proc enter-handler exit-handler [#:current-frame] [#:vm] [Scheme Procedure]
A trap that calls enter-handler when control enters proc, and exit-handler when control leaves proc.
Control can enter a procedure via:

- A procedure call.
- A return to a procedure’s frame on the stack.
- A continuation returning directly to an application of this procedure.

Control can leave a procedure via:

- A normal return from the procedure.
- An application of another procedure.
- An invocation of a continuation.
- An abort.

\[\textit{trap-instructions-in-procedure \ proc \ next-handler} \quad \text{[Scheme Procedure]}\]
\[
\text{exit-handler [#:current-frame] [#:vm]} \\
\text{A trap that calls next-handler for every instruction executed in proc, and exit-handler when execution leaves proc.}\]

\[\textit{trap-at-procedure-ip-in-range \ proc \ range \ handler} \quad \text{[Scheme Procedure]}\]
\[
\text{[#:current-frame] [#:vm]} \\
\text{A trap that calls handler when execution enters a range of instructions in proc. range is a simple of pairs, ((start . end) ...). The start addresses are inclusive, and end addresses are exclusive.}\]

\[\textit{trap-at-source-location \ file \ user-line \ handler} \quad \text{[Scheme Procedure]}\]
\[
\text{[#:current-frame] [#:vm]} \\
\text{A trap that fires when control reaches a given source location. The user-line parameter is one-indexed, as a user counts lines, instead of zero-indexed, as Guile counts lines.}\]

\[\textit{trap-frame-finish \ frame \ return-handler \ abort-handler} \quad \text{[Scheme Procedure]}\]
\[
\text{[#:vm]} \\
\text{A trap that fires when control leaves the given frame. frame should be a live frame in the current continuation. return-handler will be called on a normal return, and abort-handler on a nonlocal exit.}\]

\[\textit{trap-in-dynamic-extent \ proc \ enter-handler \ return-handler} \quad \text{[Scheme Procedure]}\]
\[
\text{abort-handler [#:vm]} \\
\text{A more traditional dynamic-wind trap, which fires enter-handler when control enters proc, return-handler on a normal return, and abort-handler on a nonlocal exit. Note that rewinds are not handled, so there is no rewind handler.}\]

\[\textit{trap-calls-in-dynamic-extent \ proc \ apply-handler} \quad \text{[Scheme Procedure]}\]
\[
\text{return-handler [#:current-frame] [#:vm]} \\
\text{A trap that calls apply-handler every time a procedure is applied, and return-handler for returns, but only during the dynamic extent of an application of proc.}\]

\[\textit{trap-instructions-in-dynamic-extent \ proc \ next-handler} \quad \text{[Scheme Procedure]}\]
\[
\text{[#:current-frame] [#:vm]} \\
\text{A trap that calls next-handler for all retired instructions within the dynamic extent of a call to proc.}\]
**trap-calls-to-procedure**  proc apply-handler                      [Scheme Procedure]

  return-handler [#:vm]

  A trap that calls apply-handler whenever proc is applied, and return-handler when it returns, but with an additional argument, the call depth.

  That is to say, the handlers will get two arguments: the frame in question, and the call depth (a non-negative integer).

**trap-matching-instructions**  frame-pred handler [#:vm]              [Scheme Procedure]

  A trap that calls frame-pred at every instruction, and if frame-pred returns a true value, calls handler on the frame.

### 6.26.4.4 Tracing Traps

The (system vm trace) module defines a number of traps for tracing of procedure applications. When a procedure is traced, it means that every call to that procedure is reported to the user during a program run. The idea is that you can mark a collection of procedures for tracing, and Guile will subsequently print out a line of the form

```
  |  |  (procedure args ...)
```

whenever a marked procedure is about to be applied to its arguments. This can help a programmer determine whether a function is being called at the wrong time or with the wrong set of arguments.

In addition, the indentation of the output is useful for demonstrating how the traced applications are or are not tail recursive with respect to each other. Thus, a trace of a non-tail recursive factorial implementation looks like this:

```
scheme@(guile-user)> (define (fact1 n)
  (if (zero? n) 1
      (* n (fact1 (1- n))))

scheme@(guile-user)> ,trace (fact1 4)
trace: (fact1 4)
trace: | (fact1 3)
trace: |   | (fact1 2)
trace: |   |   | (fact1 1)
trace: |   |   |   | (fact1 0)
trace: |   |   |   | 1
trace: |   |   |   1
trace: |   | 6
trace: 24
```

While a typical tail recursive implementation would look more like this:

```
scheme@(guile-user)> (define (facti acc n)
  (if (zero? n) acc
      (facti (* n acc) (1- n))))
scheme@(guile-user)> (define (fact2 n) (facti 1 n))
scheme@(guile-user)> ,trace (fact2 4)
trace: (fact2 4)
trace: (facti 1 4)
```
trace: (facti 4 3)
trace: (facti 12 2)
trace: (facti 24 1)
trace: (facti 24 0)
trace: 24

The low-level traps below (see Section 6.26.4.3 [Low-Level Traps], page 483) share some common options:

#:width The maximum width of trace output. Trace printouts will try not to exceed this column, but for highly nested procedure calls, it may be unavoidable. Defaults to 80.

#:vm The VM on which to add the traps. Defaults to the current thread’s VM.

#:prefix A string to print out before each trace line. As seen above in the examples, defaults to "trace: ".

To have access to these procedures, you’ll need to have imported the (system vm trace) module:

(use-modules (system vm trace))

trace-calls-to-procedure proc [#:width] [#:vm] [#:prefix] [Scheme Procedure]
Print a trace at applications of and returns from proc.

trace-calls-in-procedure proc [#:width] [#:vm] [#:prefix] [Scheme Procedure]
Print a trace at all applications and returns within the dynamic extent of calls to proc.

trace-instructions-in-procedure proc [#:width] [#:vm] [Scheme Procedure]
Print a trace at all instructions executed in the dynamic extent of calls to proc.

In addition, Guile defines a procedure to call a thunk, tracing all procedure calls and returns within the thunk.

call-with-trace thunk [#:calls?=#t] [#:instructions?=#f] [#:width=80] [Scheme Procedure]
Call thunk, tracing all execution within its dynamic extent.

If calls? is true, Guile will print a brief report at each procedure call and return, as given above.

If instructions? is true, Guile will also print a message each time an instruction is executed. This is a lot of output, but it is sometimes useful when doing low-level optimization.

Note that because this procedure manipulates the VM trace level directly, it doesn’t compose well with traps at the REPL.

See Section 4.4.4.5 [Profile Commands], page 51, for more information on tracing at the REPL.
6.26.4.5 Trap States

When multiple traps are present in a system, we begin to have a bookkeeping problem. How are they named? How does one disable, enable, or delete them?

Guile’s answer to this is to keep an implicit per-thread trap state. The trap state object is not exposed to the user; rather, API that works on trap states fetches the current trap state from the dynamic environment.

Traps are identified by integers. A trap can be enabled, disabled, or removed, and can have an associated user-visible name.

These procedures have their own module:

(use-modules (system vm trap-state))

**add-trap!** *trap name*  
Add a trap to the current trap state, associating the given name with it. Returns a fresh trap identifier (an integer).

Note that usually the more specific functions detailed in Section 6.26.4.6 [High-Level Traps], page 487, are used in preference to this one.

**list-traps**  
List the current set of traps, both enabled and disabled. Returns a list of integers.

**trap-name** *idx*  
Returns the name associated with trap *idx*, or #f if there is no such trap.

**trap-enabled?** *idx*  
Returns #t if trap *idx* is present and enabled, or #f otherwise.

**enable-trap!** *idx*  
Enables trap *idx*.

**disable-trap!** *idx*  
Disables trap *idx*.

**delete-trap!** *idx*  
Removes trap *idx*, disabling it first, if necessary.

6.26.4.6 High-Level Traps

The low-level trap API allows one to make traps that call procedures, and the trap state API allows one to keep track of what traps are there. But neither of these APIs directly helps you when you want to set a breakpoint, because it’s unclear what to do when the trap fires. Do you enter a debugger, or mail a summary of the situation to your great-aunt, or what?

So for the common case in which you just want to install breakpoints, and then have them all result in calls to one parameterizable procedure, we have the high-level trap interface.

Perhaps we should have started this section with this interface, as it’s clearly the one most people should use. But as its capabilities and limitations proceed from the lower layers, we felt that the character-building exercise of building a mental model might be helpful.

These procedures share a module with trap states:

(use-modules (system vm trap-state))
with-default-trap-handler handler thunk

Call thunk in a dynamic context in which handler is the current trap handler.

Additionally, during the execution of thunk, the VM trace level (see Section 6.26.4.1 [VM Hooks], page 481) is set to the number of enabled traps. This ensures that traps will in fact fire.

handler may be #f, in which case VM hooks are not enabled as they otherwise would be, as there is nothing to handle the traps.

The trace-level-setting behavior of with-default-trap-handler is one of its more useful aspects, but if you are willing to forgo that, and just want to install a global trap handler, there’s a function for that too:

install-trap-handler! handler

Set the current thread’s trap handler to handler.

Trap handlers are called when traps installed by procedures from this module fire. The current “consumer” of this API is Guile’s REPL, but one might easily imagine other trap handlers being used to integrate with other debugging tools.

add-trap-at-procedure-call! proc

Install a trap that will fire when proc is called.

This is a breakpoint.

add-trace-at-procedure-call! proc

Install a trap that will print a tracing message when proc is called. See Section 6.26.4.4 [Tracing Traps], page 485, for more information.

This is a tracepoint.

add-trap-at-source-location! file user-line

Install a trap that will fire when control reaches the given source location. user-line is one-indexed, as users count lines, instead of zero-indexed, as Guile counts lines.

This is a source breakpoint.

add-ephemeral-trap-at-frame-finish! frame handler

Install a trap that will call handler when frame finishes executing. The trap will be removed from the trap state after firing, or on nonlocal exit.

This is a finish trap, used to implement the “finish” REPL command.

add-ephemeral-stepping-trap! frame handler [#:into?] [#:instruction?]

Install a trap that will call handler after stepping to a different source line or instruction. The trap will be removed from the trap state after firing, or on nonlocal exit.

If instruction? is false (the default), the trap will fire when control reaches a new source line. Otherwise it will fire when control reaches a new instruction.

Additionally, if into? is false (not the default), the trap will only fire for frames at or prior to the given frame. If into? is true (the default), the trap may step into nested procedure invocations.

This is a stepping trap, used to implement the “step”, “next”, “step-instruction”, and “next-instruction” REPL commands.
6.26.5 GDB Support

Sometimes, you may find it necessary to debug Guile applications at the C level. Doing so can be tedious, in particular because the debugger is oblivious to Guile’s SCM type, and thus unable to display SCM values in any meaningful way:

```gdb
(gdb) frame
#0 scm_display (obj=0xf04310, port=0x6f9f30) at print.c:1437
```

To address that, Guile comes with an extension of the GNU Debugger (GDB) that contains a “pretty-printer” for SCM values. With this GDB extension, the C frame in the example above shows up like this:

```gdb
(gdb) frame
#0 scm_display (obj=("hello" GDB!), port=#<port file 6f9f30>) at print.c:1437
```

Here GDB was able to decode the list pointed to by `obj`, and to print it using Scheme’s read syntax.

That extension is a `.scm` file installed alongside the `libguile` shared library. When GDB 7.8 or later is installed and compiled with support for extensions written in Guile, the extension is automatically loaded when debugging a program linked against `libguile` (see Section “Auto-loading” in Debugging with GDB). Note that the directory where `libguile` is installed must be among GDB’s auto-loading “safe directories” (see Section “Auto-loading safe path” in Debugging with GDB).

6.27 Code Coverage Reports

When writing a test suite for a program or library, it is desirable to know what part of the code is covered by the test suite. The `(system vm coverage)` module provides tools to gather code coverage data and to present them, as detailed below.

**with-code-coverage thunk**  
Run `thunk`, a zero-argument procedure, while instrumenting Guile’s virtual machine to collect code coverage data. Return code coverage data and the values returned by `thunk`.

**coverage-data? obj**  
Return `#t` if `obj` is a coverage data object as returned by `with-code-coverage`.

**coverage-data->lcov data port #:key modules**  
Traverse code coverage information data, as obtained with `with-code-coverage`, and write coverage information to port in the `.info` format used by LCOV (http://ltp.sourceforge.net/coverage/lcov.php). The report will include all of `modules` (or, by default, all the currently loaded modules) even if their code was not executed.

The generated data can be fed to LCOV’s `genhtml` command to produce an HTML report, which aids coverage data visualization.

Here’s an example use:

```scheme
(use-modules (system vm coverage)
            (system vm vm))

(call-with-values (lambda ()
```
(with-code-coverage
  (lambda ()
    (do-something-tricky)))))

(lambda (data result)
  (let ((port (open-output-file "lcov.info")))
    (coverage-data-→lcov data port)
    (close file)))

In addition, the module provides low-level procedures that would make it possible to write other user interfaces to the coverage data.

_instrumented-source-files data_ [Scheme Procedures]
Return the list of “instrumented” source files, i.e., source files whose code was loaded at the time _data_ was collected.

_line-execution-counts data file_ [Scheme Procedures]
Return a list of line number/execution count pairs for _file_, or _#f_ if _file_ is not among the files covered by _data_. This includes lines with zero count.

_instrumented/executed-lines data file_ [Scheme Procedures]
Return the number of instrumented and the number of executed source lines in _file_ according to _data_.

_procedure-execution-count data proc_ [Scheme Procedures]
Return the number of times _proc’s_ code was executed, according to _data_, or _#f_ if _proc_ was not executed. When _proc_ is a closure, the number of times its code was executed is returned, not the number of times this code associated with this particular closure was executed.
7 Guile Modules

7.1 SLIB

SLIB is a portable library of Scheme packages which can be used with Guile and other Scheme implementations. SLIB is not included in the Guile distribution, but can be installed separately (see Section 7.1.1 [SLIB installation], page 491). It is available from http://people.csail.mit.edu/jaffer/SLIB.html.

After SLIB is installed, the following Scheme expression must be executed before the SLIB facilities can be used:

```scheme
(use-modules (ice-9 slib))
```

`require` can then be used in the usual way (see Section “Require” in The SLIB Manual). For example,

```scheme
(use-modules (ice-9 slib))
(require 'primes)
(prime? 13)
⇒ #t
```

A few Guile core functions are overridden by the SLIB setups; for example the SLIB version of `delete-file` returns a boolean indicating success or failure, whereas the Guile core version throws an error for failure. In general (and as might be expected) when SLIB is loaded it’s the SLIB specifications that are followed.

7.1.1 SLIB installation

The following procedure works, e.g., with SLIB version 3a3 (see Section “Installation” in The SLIB Portable Scheme Library):

1. Unpack SLIB and install it using `make install` from its directory. By default, this will install SLIB in `/usr/local/lib/slib/`. Running `make install-info` installs its documentation, by default under `/usr/local/info/`.

2. Define the `SCHEME_LIBRARY_PATH` environment variable:

   ```bash
   $ SCHEME_LIBRARY_PATH=/usr/local/lib/slib/
   $ export SCHEME_LIBRARY_PATH
   ```

   Alternatively, you can create a symlink in the Guile directory to SLIB, e.g.:

   ```bash
   ln -s /usr/local/lib/slib /usr/local/share/guile/2.2/slib
   ```

3. Use Guile to create the catalog file, e.g.:

   ```bash
   # guile
   guile> (use-modules (ice-9 slib))
   guile> (require 'new-catalog)
   guile> (quit)
   ```

   The catalog data should now be in `/usr/local/share/guile/2.2/slibcat`.

   If instead you get an error such as:

   ```bash
   Unbound variable: scheme-implementation-type
   ```

   then a solution is to get a newer version of Guile, or to modify `ice-9/slib.scm` to use `define-public` for the offending variables.
7.1.2 JACAL

Jacal is a symbolic math package written in Scheme by Aubrey Jaffer. It is usually installed as an extra package in SLIB.

You can use Guile's interface to SLIB to invoke Jacal:

```scheme
(use-modules (ice-9 slib))
(slib:load "math")
(math)
```

For complete documentation on Jacal, please read the Jacal manual. If it has been installed on line, you can look at Section “Jacal” in JACAL Symbolic Mathematics System. Otherwise you can find it on the web at http://www-swiss.ai.mit.edu/~jaffer/JACAL.html.

7.2 POSIX System Calls and Networking

7.2.1 POSIX Interface Conventions

These interfaces provide access to operating system facilities. They provide a simple wrapping around the underlying C interfaces to make usage from Scheme more convenient. They are also used to implement the Guile port of scsh (see Section 7.17 [The Scheme shell (scsh)], page 728).

Generally there is a single procedure for each corresponding Unix facility. There are some exceptions, such as procedures implemented for speed and convenience in Scheme with no primitive Unix equivalent, e.g., `copy-file`.

The interfaces are intended as far as possible to be portable across different versions of Unix. In some cases procedures which can't be implemented on particular systems may become no-ops, or perform limited actions. In other cases they may throw errors.

General naming conventions are as follows:

- The Scheme name is often identical to the name of the underlying Unix facility.
- Underscores in Unix procedure names are converted to hyphens.
- Procedures which destructively modify Scheme data have exclamation marks appended, e.g., `recv!`.
- Predicates (returning only `#t` or `#f`) have question marks appended, e.g., `access?`.
- Some names are changed to avoid conflict with dissimilar interfaces defined by scsh, e.g., `primitive-fork`.
- Unix preprocessor names such as `EPERM` or `R_OK` are converted to Scheme variables of the same name (underscores are not replaced with hyphens).

Unexpected conditions are generally handled by raising exceptions. There are a few procedures which return a special value if they don't succeed, e.g., `getenv` returns `#f` if it the requested string is not found in the environment. These cases are noted in the documentation.

For ways to deal with exceptions, see Section 6.13.8 [Exceptions], page 311.

Errors which the C library would report by returning a null pointer or through some other means are reported by raising a `system-error` exception with `scm-error` (see Section 6.13.9...
The `data` parameter is a list containing the Unix `errno` value (an integer). For example,

```
(define (my-handler key func fmt fmtargs data)
  (display key) (newline)
  (display func) (newline)
  (apply format #t fmt fmtargs) (newline)
  (display data) (newline))
```

```
(catch 'system-error
  (lambda () (dup2 -123 -456))
  my-handler)
```

- `system-error`
- `dup2`
- Bad file descriptor
- (9)

**system-error-errno arglist**  
Return the `errno` value from a list which is the arguments to an exception handler. If the exception is not a `system-error`, then the return is `#f`. For example,

```
(catch 'system-error
  (lambda () (mkdir "/this-ought-to-fail-if-I’m-not-root"))
  (lambda stuff
    (let ((errno (system-error-errno stuff)))
      (cond
        (= errno EACCES)
        (display "You’re not allowed to do that.")
        (= errno EEXIST)
        (display "Already exists.")
        (#t
          (display (strerror errno)))))
  (newline))
```

### 7.2.2 Ports and File Descriptors

Conventions generally follow those of scons, Section 7.17 [The Scheme shell (scons)], page 728.

Each open file port has an associated operating system file descriptor. File descriptors are generally not useful in Scheme programs; however they may be needed when interfacing with foreign code and the Unix environment.

A file descriptor can be extracted from a port and a new port can be created from a file descriptor. However a file descriptor is just an integer and the garbage collector doesn’t recognize it as a reference to the port. If all other references to the port were dropped, then it’s likely that the garbage collector would free the port, with the side-effect of closing the file descriptor prematurely.
To assist the programmer in avoiding this problem, each port has an associated \textit{revealed count} which can be used to keep track of how many times the underlying file descriptor has been stored in other places. If a port’s revealed count is greater than zero, the file descriptor will not be closed when the port is garbage collected. A programmer can therefore ensure that the revealed count will be greater than zero if the file descriptor is needed elsewhere.

For the simple case where a file descriptor is “imported” once to become a port, it does not matter if the file descriptor is closed when the port is garbage collected. There is no need to maintain a revealed count. Likewise when “exporting” a file descriptor to the external environment, setting the revealed count is not required provided the port is kept open (i.e., is pointed to by a live Scheme binding) while the file descriptor is in use.

To correspond with traditional Unix behaviour, three file descriptors (0, 1, and 2) are automatically imported when a program starts up and assigned to the initial values of the current/standard input, output, and error ports, respectively. The revealed count for each is initially set to one, so that dropping references to one of these ports will not result in its garbage collection: it could be retrieved with \texttt{fdopen} or \texttt{fdes->ports}.

Guile’s ports can be buffered. This means that writing a byte to a file port goes to the internal buffer first, and only when the buffer is full (or the user invokes \texttt{force-output} on the port) is the data actually written to the file descriptor. Likewise on input, bytes are read in from the file descriptor in blocks and placed in a buffer. Reading a character via \texttt{read-char} first goes to the buffer, filling it as needed. Usually read buffering is more or less transparent, but write buffering can sometimes cause writes to be delayed unexpectedly, if you forget to call \texttt{force-output}. See Section 6.14.6 [Buffering], page 336, for more on how to control port buffers.

Note however that some procedures (e.g., \texttt{recv!}) will accept ports as arguments, but will actually operate directly on the file descriptor underlying the port. Any port buffering is ignored, including the buffer which implements \texttt{peek-char} and \texttt{unread-char}.

\begin{verbatim}
port-revealed port               [Scheme Procedure]
scm_port_revealed (port)         [C Function]
    Return the revealed count for \texttt{port}.

set-port-revealed! port rcount   [Scheme Procedure]
scm_set_port_revealed_x (port, rcount)  [C Function]
    Sets the revealed count for a \texttt{port} to \texttt{rcount}. The return value is unspecified.

fileno port                      [Scheme Procedure]
scm_fileno (port)               [C Function]
    Return the integer file descriptor underlying \texttt{port}. Does not change its revealed count.

port->fdes port                  [Scheme Procedure]
    Returns the integer file descriptor underlying \texttt{port}. As a side effect the revealed count of \texttt{port} is incremented.

fdopen fdes modes                [Scheme Procedure]
scm_fdopen (fdes, modes)         [C Function]
    Return a new port based on the file descriptor \texttt{fdes}. Modes are given by the string \texttt{modes}. The revealed count of the port is initialized to zero. The \texttt{modes} string is the same as that accepted by \texttt{open-file} (see Section 6.14.10.1 [File Ports], page 341).
\end{verbatim}
Chapter 7: Guile Modules

**fdes->ports fdes**  
[Scheme Procedure]

**scm_fdes_to_ports (fdes)**  
[C Function]

Return a list of existing ports which have `fdes` as an underlying file descriptor, without changing their revealed counts.

**fdes->inport fdes**  
[Scheme Procedure]

Returns an existing input port which has `fdes` as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new input port with a revealed count of 1.

**fdes->outport fdes**  
[Scheme Procedure]

Returns an existing output port which has `fdes` as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new output port with a revealed count of 1.

**primitive-move->fdes port fdes**  
[Scheme Procedure]

**scm_primitive_move_to_fdes (port, fdes)**  
[C Function]

Moves the underlying file descriptor for `port` to the integer value `fdes` without changing the revealed count of `port`. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is `#f` if the file descriptor already had the required value or `#t` if it was moved.

**move->fdes port fdes**  
[Scheme Procedure]

Moves the underlying file descriptor for `port` to the integer value `fdes` and sets its revealed count to one. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is unspecified.

**release-port-handle port**  
[Scheme Procedure]

Decrement the revealed count for a port.

**fsync port_or_fd**  
[Scheme Procedure]

**scm_fsync (port_or_fd)**  
[C Function]

Copies any unwritten data for the specified output file descriptor to disk. If `port_or_fd` is a port, its buffer is flushed before the underlying file descriptor is fsync’d. The return value is unspecified.

**open path flags [mode]**  
[Scheme Procedure]

**scm_open (path, flags, mode)**  
[C Function]

Open the file named by `path` for reading and/or writing. `flags` is an integer specifying how the file should be opened. `mode` is an integer specifying the permission bits of the file, if it needs to be created, before the umask (see Section 7.2.7 [Processes], page 514) is applied. The default is 666 (Unix itself has no default).

`flags` can be constructed by combining variables using `logior`. Basic flags are:

**O_RDONLY**  
[Variable]

Open the file read-only.
O_WRONLY
  Open the file write-only.

O_RDWR
  Open the file read/write.

O_APPEND
  Append to the file instead of truncating.

O_CREAT
  Create the file if it does not already exist.


open-fdes path flags [mode]
  Scheme Procedure

scm_open_fdes (path, flags, mode)
  C Function

Similar to open but return a file descriptor instead of a port.

close fd_or_port
  Scheme Procedure

scm_close (fd_or_port)
  C Function

Similar to close-port (see Section 6.14.1 [Ports], page 328), but also works on file descriptors. A side effect of closing a file descriptor is that any ports using that file descriptor are moved to a different file descriptor and have their revealed counts set to zero.

close-fdes fd
  Scheme Procedure

scm_close_fdes (fd)
  C Function

A simple wrapper for the close system call. Close file descriptor fd, which must be an integer. Unlike close, the file descriptor will be closed even if a port is using it. The return value is unspecified.

pipe
  Scheme Procedure

scm_pipe ()
  C Function

Return a newly created pipe: a pair of ports which are linked together on the local machine. The CAR is the input port and the CDR is the output port. Data written (and flushed) to the output port can be read from the input port. Pipes are commonly used for communication with a newly forked child process. The need to flush the output port can be avoided by making it unbuffered using setvbuf (see Section 6.14.6 [Buffering], page 336).

PIPE_BUF
  Variable

A write of up to PIPE_BUF many bytes to a pipe is atomic, meaning when done it goes into the pipe instantaneously and as a contiguous block (see Section “Atomicity of Pipe I/O” in The GNU C Library Reference Manual).

Note that the output port is likely to block if too much data has been written but not yet read from the input port. Typically the capacity is PIPE_BUF bytes.
The next group of procedures perform a \texttt{dup2} system call, if \texttt{newfd} (an integer) is supplied, otherwise a \texttt{dup}. The file descriptor to be duplicated can be supplied as an integer or contained in a port. The type of value returned varies depending on which procedure is used.

All procedures also have the side effect when performing \texttt{dup2} that any ports using \texttt{newfd} are moved to a different file descriptor and have their revealed counts set to zero.

\texttt{dup->fdes fd\_or\_port \[fd]} \hspace{1cm} \text{[Scheme Procedure]}
\texttt{scm\_dup\_to\_fdes (fd\_or\_port, \textit{fd})} \hspace{1cm} \text{[C Function]}
Return a new integer file descriptor referring to the open file designated by \textit{fd\_or\_port}, which must be either an open file port or a file descriptor.

\texttt{dup->inport port/fd \[newfd]} \hspace{1cm} \text{[Scheme Procedure]}
Returns a new input port using the new file descriptor.

\texttt{dup->outport port/fd \[newfd]} \hspace{1cm} \text{[Scheme Procedure]}
Returns a new output port using the new file descriptor.

\texttt{dup port/fd \[newfd]} \hspace{1cm} \text{[Scheme Procedure]}
Returns a new port if \textit{port/fd} is a port, with the same mode as the supplied port, otherwise returns an integer file descriptor.

\texttt{dup->port port/fd mode \[newfd]} \hspace{1cm} \text{[Scheme Procedure]}
Returns a new port using the new file descriptor. \textit{mode} supplies a mode string for the port (see Section 6.14.10.1 [File Ports], page 341).

\texttt{duplicate-port port modes} \hspace{1cm} \text{[Scheme Procedure]}
Returns a new port which is opened on a duplicate of the file descriptor underlying \textit{port}, with mode string \textit{modes} as for Section 6.14.10.1 [File Ports], page 341. The two ports will share a file position and file status flags. Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered. The mode string can include 0 to obtain an unbuffered duplicate port.
This procedure is equivalent to (\texttt{dup->port port modes}).

\texttt{redirect-port old\_port new\_port} \hspace{1cm} \text{[Scheme Procedure]}
\texttt{scm\_redirect\_port (old\_port, new\_port)} \hspace{1cm} \text{[C Function]}
This procedure takes two ports and duplicates the underlying file descriptor from \textit{old\_port} into \textit{new\_port}. The current file descriptor in \textit{new\_port} will be closed. After the redirection the two ports will share a file position and file status flags.
The return value is unspecified.
Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered.
This procedure does not have any side effects on other ports or revealed counts.

\texttt{dup2 oldfd newfd} \hspace{1cm} \text{[Scheme Procedure]}
\texttt{scm\_dup2 (oldfd, newfd)} \hspace{1cm} \text{[C Function]}
A simple wrapper for the \texttt{dup2} system call. Copies the file descriptor \texttt{oldfd} to descriptor number \texttt{newfd}, replacing the previous meaning of \texttt{newfd}. Both \texttt{oldfd} and \texttt{newfd}
must be integers. Unlike for `dup->fdes` or `primitive-move->fdes`, no attempt is made to move away ports which are using `newfd`. The return value is unspecified.

### port-for-each proc

[Scheme Procedure]

`scm_port_for_each (SCM proc)`

[Function]

Apply `proc` to each port in the Guile port table (FIXME: what is the Guile port table?) in turn. The return value is unspecified. More specifically, `proc` is applied exactly once to every port that exists in the system at the time `port-for-each` is invoked. Changes to the port table while `port-for-each` is running have no effect as far as `port-for-each` is concerned.

The C function `scm_port_for_each` takes a Scheme procedure encoded as a SCM value, while `scm_c_port_for_each` takes a pointer to a C function and passes along a arbitrary data cookie.

### fcntl port/fd cmd [value]

[Scheme Procedure]

`scm_fcntl (object, cmd, value)`

[Function]

Apply `cmd` on `port/fd`, either a port or file descriptor. The `value` argument is used by the SET commands described below, it’s an integer value.

Values for `cmd` are:

- **F_DUPFD**
  Duplicate the file descriptor, the same as `dup->fdes` above does.

- **F_GETFD**
  Get or set flags associated with the file descriptor. The only flag is the following,

- **F_SETFD**
  Get or set flags associated with the file descriptor. The only flag is the following,

- **FD_CLOEXEC**
  “Close on exec”, meaning the file descriptor will be closed on an exec call (a successful such call). For example to set that flag,

  ```lisp
  (fcntl port F_SETFD FD_CLOEXEC)
  ```

  Or better, set it but leave any other possible future flags unchanged,

  ```lisp
  (fcntl port F_SETFD (logior FD_CLOEXEC (fcntl port F_GETFD)))
  ```

- **F_GETFL**
  Get or set flags associated with the open file. These flags are O_RDONLY etc described under `open` above.

  A common use is to set O_NONBLOCK on a network socket. The following sets that flag, and leaves other flags unchanged.

  ```lisp
  (fcntl sock F_SETFL (logior O_NONBLOCK (fcntl sock F_GETFL)))
  ```

- **F_GETOWN**
  Get or set the process ID of a socket’s owner, for SIGIO signals.
flock file operation

**[Scheme Procedure]**

**scm_flock** (file, operation)

Apply or remove an advisory lock on an open file. operation specifies the action to be done:

**LOCK_SH**

[Variable]

Shared lock. More than one process may hold a shared lock for a given file at a given time.

**LOCK_EX**

[Variable]

Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

**LOCK_UN**

Unlock the file.

**LOCK_NB**

[Variable]

Don’t block when locking. This is combined with one of the other operations using logior (see Section 6.6.2.13 [Bitwise Operations], page 125). If flock would block an EWOULDBLOCK error is thrown (see Section 7.2.1 [Conventions], page 492).

The return value is not specified. file may be an open file descriptor or an open file descriptor port.

Note that flock does not lock files across NFS.

**select** reads writes excepts [secs [usecs]]

**[Scheme Procedure]**

**scm_select** (reads, writes, excepts, secs, usecs)

**[C Function]**

This procedure has a variety of uses: waiting for the ability to provide input, accept output, or the existence of exceptional conditions on a collection of ports or file descriptors, or waiting for a timeout to occur.

When an error occurs, this procedure throws a **system-error** exception (see Section 7.2.1 [Conventions], page 492). Note that select may return early for other reasons, for example due to pending interrupts. See Section 6.22.3 [Asyncs], page 439, for more on interrupts.

reads, writes and excepts can be lists or vectors, with each member a port or a file descriptor. The value returned is a list of three corresponding lists or vectors containing only the members which meet the specified requirement. The ability of port buffers to provide input or accept output is taken into account. Ordering of the input lists or vectors is not preserved.

The optional arguments secs and usecs specify the timeout. Either secs can be specified alone, as either an integer or a real number, or both secs and usecs can be specified as integers, in which case usecs is an additional timeout expressed in microseconds. If secs is omitted or is #f then select will wait for as long as it takes for one of the other conditions to be satisfied.

The scsh version of select differs as follows: Only vectors are accepted for the first three arguments. The usecs argument is not supported. Multiple values are returned instead of a list. Duplicates in the input vectors appear only once in output. An additional select! interface is provided.
While it is sometimes necessary to operate at the level of file descriptors, this is an operation whose correctness can only be considered as part of a whole program. So for example while the effects of `(string-set! x 34 #\y)` are limited to the bits of code that can access x, `(close-fdes 34)` mutates the state of the entire process. In particular if another thread is using file descriptor 34 then their state might be corrupted; and another thread which opens a file might cause file descriptor 34 to be re-used, so that corruption could manifest itself in a strange way.

However when working with file descriptors, it’s common to want to associate information with the file descriptor, perhaps in a side table. To support this use case and to allow user code to remove an association when a file descriptor is closed, Guile offers `fdes finalizers`.

As the name indicates, fdes finalizers are finalizers – they can run in response to garbage collection, and they can also run in response to explicit calls to `close-port`, `close-fdes`, or the like. As such they inherit many of the pitfalls of finalizers: they may be invoked from concurrent threads, or not at all. See Section 5.5.4 [Foreign Object Memory Management], page 77, for more on finalizers.

To use fdes finalizers, import their module;

```
(use-modules (ice-9 fdes-finalizers))
```

`add-fdes-finalizer! fdes finalizer` [Scheme Procedure]

Add or remove a finalizer for `fdes`. A finalizer is a procedure that is called by Guile when a file descriptor is closed. The file descriptor being closed is passed as the one argument to the finalizer. If a finalizer has been added multiple times to a file descriptor, to remove it would require that number of calls to `remove-fdes-finalizer!`.

The finalizers added to a file descriptor are called by Guile in an unspecified order, and their return values are ignored.

### 7.2.3 File System

These procedures allow querying and setting file system attributes (such as owner, permissions, sizes and types of files); deleting, copying, renaming and linking files; creating and removing directories and querying their contents; syncing the file system and creating special files.

`access? path how` [Scheme Procedure]

`scm_access (path, how)` [C Function]

Test accessibility of a file under the real UID and GID of the calling process. The return is `#t` if `path` exists and the permissions requested by `how` are all allowed, or `#f` if not.

`how` is an integer which is one of the following values, or a bitwise-OR (`logior`) of multiple values.

- `R_OK` [Variable]
  Test for read permission.

- `W_OK` [Variable]
  Test for write permission.
X_OK

Test for execute permission.

F_OK

Test for existence of the file. This is implied by each of the other tests, so there’s no need to combine it with them.

It’s important to note that **access** does not simply indicate what will happen on attempting to read or write a file. In normal circumstances it does, but in a set-UID or set-GID program it doesn’t because **access** tests the real ID, whereas an open or execute attempt uses the effective ID.

A program which will never run set-UID/GID can ignore the difference between real and effective IDs, but for maximum generality, especially in library functions, it’s best not to use **access** to predict the result of an open or execute, instead simply attempt that and catch any exception.

The main use for **access** is to let a set-UID/GID program determine what the invoking user would have been allowed to do, without the greater (or perhaps lesser) privileges afforded by the effective ID. For more on this, see Section “Testing File Access” in *The GNU C Library Reference Manual*.

### Scheme Procedure

`stat object [exception-on-error?]`

Return an object containing various information about the file determined by `object`. `object` can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fstat` is used as the underlying system call).

If the optional `exception-on-error` argument is true, which is the default, an exception will be raised if the underlying system call returns an error, for example if the file is not found or is not readable. Otherwise, an error will cause `stat` to return `#f`.

The object returned by `stat` can be passed as a single parameter to the following procedures, all of which return integers:

### Scheme Procedure

`stat:dev st`

The device number containing the file.

`stat:ino st`

The file serial number, which distinguishes this file from all other files on the same device.

`stat:mode st`

The mode of the file. This is an integer which incorporates file type information and file permission bits. See also `stat:type` and `stat:perms` below.

`stat:nlink st`

The number of hard links to the file.

`stat:uid st`

The user ID of the file’s owner.

`stat:gid st`

The group ID of the file.
**stat:rdev** *st*  
[Scheme Procedure]  
Device ID; this entry is defined only for character or block special files. On some systems this field is not available at all, in which case **stat:rdev** returns #f.

**stat:size** *st*  
[Scheme Procedure]  
The size of a regular file in bytes.

**stat:atime** *st*  
[Scheme Procedure]  
The last access time for the file, in seconds.

**stat:mtime** *st*  
[Scheme Procedure]  
The last modification time for the file, in seconds.

**stat:ctime** *st*  
[Scheme Procedure]  
The last modification time for the attributes of the file, in seconds.

**stat:atimensec** *st*  
[Scheme Procedure]  
**stat:mtimensec** *st*  
[Scheme Procedure]  
**stat:ctimensec** *st*  
The fractional part of a file’s access, modification, or attribute modification time, in nanoseconds. Nanosecond timestamps are only available on some operating systems and file systems. If Guile cannot retrieve nanosecond-level timestamps for a file, these fields will be set to 0.

**stat:blksize** *st*  
[Scheme Procedure]  
The optimal block size for reading or writing the file, in bytes. On some systems this field is not available, in which case **stat:blksize** returns a sensible suggested block size.

**stat:blocks** *st*  
[Scheme Procedure]  
The amount of disk space that the file occupies measured in units of 512 byte blocks. On some systems this field is not available, in which case **stat:blocks** returns #f.

In addition, the following procedures return the information from **stat:mode** in a more convenient form:

**stat:type** *st*  
[Scheme Procedure]  
A symbol representing the type of file. Possible values are ’regular’, ’directory’, ’symlink’, ’block-special’, ’char-special’, ’fifo’, ’socket’, and ’unknown’.

**stat:perms** *st*  
[Scheme Procedure]  
An integer representing the access permission bits.

**lstat** *path*  
[Scheme Procedure]  
**scm_lstat** *(path)*  
[C Function]  
Similar to **stat**, but does not follow symbolic links, i.e., it will return information about a symbolic link itself, not the file it points to. *path* must be a string.
**readlink** *path*  
**s**cm_readlink (*path*)  
Return the value of the symbolic link named by *path* (a string), i.e., the file that the link points to.

**chown** *object owner group*  
**s**cm_chown (*object, owner, group*)  
Change the ownership and group of the file referred to by *object* to the integer values *owner* and *group*. *object* can be a string containing a file name or, if the platform supports fchown (see Section “File Owner” in The GNU C Library Reference Manual), a port or integer file descriptor which is open on the file. The return value is unspecified.

If *object* is a symbolic link, either the ownership of the link or the ownership of the referenced file will be changed depending on the operating system (lchown is unsupported at present). If *owner* or *group* is specified as -1, then that ID is not changed.

**chmod** *object mode*  
**s**cm_chmod (*object, mode*)  
Changes the permissions of the file referred to by *object*. *object* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case fchmod is used as the underlying system call). *mode* specifies the new permissions as a decimal number, e.g., `(chmod "foo" #o755)`. The return value is unspecified.

**utime** *pathname* [actime [modtime [actimens [modtimens [flags]]]]]]  
**s**cm_utime (*pathname, actime, modtime, actimens, modtimens, flags*)  
*utime* sets the access and modification times for the file named by *pathname*. If *actime* or *modtime* is not supplied, then the current time is used. *actime* and *modtime* must be integer time values as returned by the current-time procedure.

The optional *actimens* and *modtimens* are nanoseconds to add *actime* and *modtime*. Nanosecond precision is only supported on some combinations of file systems and operating systems.

```
    (utime "foo" (- (current-time) 3600))
```

will set the access time to one hour in the past and the modification time to the current time.

Last, *flags* may be either 0 or the AT_SYMLINK_NOFOLLOW constant, to set the time of *pathname* even if it is a symbolic link.

**delete-file** *str*  
**s**cm_delete_file (*str*)  
Deletes (or “unlinks”) the file whose path is specified by *str*.

**copy-file** *oldfile newfile*  
**s**cm_copy_file (*oldfile, newfile*)  
Copy the file specified by *oldfile* to *newfile*. The return value is unspecified.
sendfile out in count [offset]  

sendfile (out, in, count, offset)  

Send count bytes from in to out, both of which must be either open file ports or file descriptors. When offset is omitted, start reading from in's current position; otherwise, start reading at offset. Return the number of bytes actually sent.

When in is a port, it is often preferable to specify offset, because in’s offset as a port may be different from the offset of its underlying file descriptor.

On systems that support it, such as GNU/Linux, this procedure uses the sendfile libc function, which usually corresponds to a system call. This is faster than doing a series of read and write system calls. A typical application is to send a file over a socket.

In some cases, the sendfile libc function may return EINVAL or ENOSYS. In that case, Guile’s sendfile procedure automatically falls back to doing a series of read and write calls.

In other cases, the libc function may send fewer bytes than count—for instance because out is a slow or limited device, such as a pipe. When that happens, Guile’s sendfile automatically retries until exactly count bytes were sent or an error occurs.

rename-file oldname newname  

rename (oldname, newname)  

Renames the file specified by oldname to newname. The return value is unspecified.

link oldpath newpath  

link (oldpath, newpath)  

Creates a new name newpath in the file system for the file named by oldpath. If oldpath is a symbolic link, the link may or may not be followed depending on the system.

symlink oldpath newpath  

symlink (oldpath, newpath)  

Create a symbolic link named newpath with the value (i.e., pointing to) oldpath. The return value is unspecified.

mkdir path [mode]  

mkdir (path, mode)  

Create a new directory named by path. If mode is omitted then the permissions of the directory are set to #o777 masked with the current umask (see Section 7.2.7 [Processes], page 514). Otherwise they are set to the value specified with mode. The return value is unspecified.

rmdir path  

rmdir (path)  

Remove the existing directory named by path. The directory must be empty for this to succeed. The return value is unspecified.

opendir dirname  

opendir (dirname)  

Open the directory specified by dirname and return a directory stream.
Before using this and the procedures below, make sure to see the higher-level procedures for directory traversal that are available (see Section 7.11 [File Tree Walk], page 711).

**directory-stream? object**  
[Scheme Procedure]

**scm_directory_stream_p (object)**  
[C Function]

Return a boolean indicating whether object is a directory stream as returned by opendir.

**readdir stream**  
[Scheme Procedure]

**scm_readdir (stream)**  
[C Function]

Return (as a string) the next directory entry from the directory stream stream. If there is no remaining entry to be read then the end of file object is returned.

**rewinddir stream**  
[Scheme Procedure]

**scm_rewinddir (stream)**  
[C Function]

Reset the directory port stream so that the next call to readdir will return the first directory entry.

**closedir stream**  
[Scheme Procedure]

**scm_closedir (stream)**  
[C Function]

Close the directory stream stream. The return value is unspecified.

Here is an example showing how to display all the entries in a directory:

```
(define dir (opendir "/usr/lib"))
(do ((entry (readdir dir) (readdir dir)))
   ((eof-object? entry))
   (display entry)(newline))
(closedir dir)
```

**sync**  
[Scheme Procedure]

**scm_sync ()**  
[C Function]

Flush the operating system disk buffers. The return value is unspecified.

**mknod path type perms dev**  
[Scheme Procedure]

**scm_mknod (path, type, perms, dev)**  
[C Function]

Creates a new special file, such as a file corresponding to a device. path specifies the name of the file. type should be one of the following symbols: 'regular', 'directory', 'symlink', 'block-special', 'char-special', 'fifo', or 'socket'. perms (an integer) specifies the file permissions. dev (an integer) specifies which device the special file refers to. Its exact interpretation depends on the kind of special file being created.

E.g.,

```
(mknod "/dev/fd0" 'block-special #o660 (+ (* 2 256) 2))
```

The return value is unspecified.

**tmpnam**  
[Scheme Procedure]

**scm_tmpnam ()**  
[C Function]

Return an auto-generated name of a temporary file, a file which doesn’t already exist. The name includes a path, it’s usually in /tmp but that’s system dependent.
Care must be taken when using \texttt{tmpnam}. In between choosing the name and creating the file another program might use that name, or an attacker might even make it a symlink pointing at something important and causing you to overwrite that.

The safe way is to create the file using \texttt{open} with \texttt{O_EXCL} to avoid any overwriting. A loop can try again with another name if the file exists (error \texttt{EEXIST}). \texttt{mkstemp!} below does that.

\begin{verbatim}
mkstemp!tmpl [mode] \hspace{2cm} \text{[Scheme Procedure]}
scm_mkstemp (tmpl) \hspace{2cm} \text{[C Function]}
\end{verbatim}

Create a new unique file in the file system and return a new buffered port open for reading and writing to the file.

\textit{tmpl} is a string specifying where the file should be created: it must end with ‘XXXXXX’ and those ‘X’s will be changed in the string to return the name of the file.

(\textit{port-filename} on the port also gives the name.)

POSIX doesn’t specify the permissions mode of the file, on GNU and most systems it’s \#o600. An application can use \texttt{chmod} to relax that if desired. For example \#o666 less umask, which is usual for ordinary file creation,

\begin{verbatim}
(let ((port (mkstemp! (string-copy "/tmp/myfile-XXXXXX"))))
  (chmod port (logand \#o666 (lognot (umask))))
  ...)
\end{verbatim}

The optional \textit{mode} argument specifies a mode with which to open the new file, as a string in the same format that \texttt{open-file} takes. It defaults to "w+".

\begin{verbatim}
tmpfile \hspace{2cm} \text{[Scheme Procedure]}
scm_tmpfile () \hspace{2cm} \text{[C Function]}
\end{verbatim}

Return an input/output port to a unique temporary file named using the path prefix \texttt{P_tmpdir} defined in \texttt{stdio.h}. The file is automatically deleted when the port is closed or the program terminates.

\begin{verbatim}
dirname filename \hspace{2cm} \text{[Scheme Procedure]}
scm_dirname (filename) \hspace{2cm} \text{[C Function]}
\end{verbatim}

Return the directory name component of the file name \texttt{filename}. If \texttt{filename} does not contain a directory component, . is returned.

\begin{verbatim}
basename filename [suffix] \hspace{2cm} \text{[Scheme Procedure]}
scm_basename (filename, suffix) \hspace{2cm} \text{[C Function]}
\end{verbatim}

Return the base name of the file name \texttt{filename}. The base name is the file name without any directory components. If \texttt{suffix} is provided, and is equal to the end of \texttt{basename}, it is removed also.

\begin{verbatim}
(basename "/tmp/test.xml" ".xml")
\Rightarrow "test"
\end{verbatim}

\begin{verbatim}
file-exists? filename \hspace{2cm} \text{[Scheme Procedure]}
\end{verbatim}

Return \#t if the file named \texttt{filename} exists, \#f if not.

Many operating systems, such as GNU, use / (forward slash) to separate the components of a file name; any file name starting with / is considered an \texttt{absolute file name}. These
conventions are specified by the POSIX Base Definitions, which refer to conforming file names as “pathnames”. Some operating systems use a different convention; in particular, Windows uses \ (backslash) as the file name separator, and also has the notion of volume names like C:\ for absolute file names. The following procedures and variables provide support for portable file name manipulations.

```scheme
system-file-name-convention [Scheme Procedure]
  Return either posix or windows, depending on what kind of system this Guile is running on.
```

```scheme
file-name-separator? c [Scheme Procedure]
  Return true if character c is a file name separator on the host platform.
```

```scheme
absolute-file-name? file-name [Scheme Procedure]
  Return true if file-name denotes an absolute file name on the host platform.
```

```scheme
file-name-separator-string [Scheme Variable]
  The preferred file name separator.
  Note that on MinGW builds for Windows, both / and \ are valid separators. Thus, programs should not assume that file-name-separator-string is the only file name separator—e.g., when extracting the components of a file name.
```

### 7.2.4 User Information

The facilities in this section provide an interface to the user and group database. They should be used with care since they are not reentrant.

The following functions accept an object representing user information and return a selected component:

```scheme
passwd:name pw [Scheme Procedure]
  The name of the userid.
```

```scheme
passwd:passwd pw [Scheme Procedure]
  The encrypted passwd.
```

```scheme
passwd:uid pw [Scheme Procedure]
  The user id number.
```

```scheme
passwd:gid pw [Scheme Procedure]
  The group id number.
```

```scheme
passwd:gecos pw [Scheme Procedure]
  The full name.
```

```scheme
passwd:dir pw [Scheme Procedure]
  The home directory.
```

```scheme
passwd:shell pw [Scheme Procedure]
  The login shell.
getpwuid \textit{uid} \hspace{1cm} \textbf{[Scheme Procedure]}
Look up an integer userid in the user database.

getpwnam \textit{name} \hspace{1cm} \textbf{[Scheme Procedure]}
Look up a user name string in the user database.

\textbf{setpwent} \hspace{1cm} \textbf{[Scheme Procedure]}
Initializes a stream used by \texttt{getpwent} to read from the user database. The next use of \texttt{getpwent} will return the first entry. The return value is unspecified.

\textbf{getpwent} \hspace{1cm} \textbf{[Scheme Procedure]}
Read the next entry in the user database stream. The return is a passwd user object as above, or \texttt{#f} when no more entries.

\textbf{endpwent} \hspace{1cm} \textbf{[Scheme Procedure]}
Closes the stream used by \texttt{getpwent}. The return value is unspecified.

\textbf{setpw} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{scm\_setpwent} \hspace{1cm} \textbf{[C Function]}
If called with a true argument, initialize or reset the password data stream. Otherwise, close the stream. The \texttt{setpwent} and \texttt{endpwent} procedures are implemented on top of this.

\textbf{getpw} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{scm\_getpwuid} \hspace{1cm} \textbf{[C Function]}
Look up an entry in the user database. \texttt{user} can be an integer, a string, or omitted, giving the behaviour of getpwuid, getpwnam or getpwent respectively.

The following functions accept an object representing group information and return a selected component:

\textbf{group\_name} \hspace{1cm} \textbf{[Scheme Procedure]}
The group name.

\textbf{group\_passwd} \hspace{1cm} \textbf{[Scheme Procedure]}
The encrypted group password.

\textbf{group\_gid} \hspace{1cm} \textbf{[Scheme Procedure]}
The group id number.

\textbf{group\_mem} \hspace{1cm} \textbf{[Scheme Procedure]}
A list of userids which have this group as a supplementary group.

\textbf{getgrgid} \hspace{1cm} \textbf{[Scheme Procedure]}
Look up an integer group id in the group database.

\textbf{getgrnam} \hspace{1cm} \textbf{[Scheme Procedure]}
Look up a group name in the group database.

\textbf{setgrent} \hspace{1cm} \textbf{[Scheme Procedure]}
Initializes a stream used by \texttt{getgrent} to read from the group database. The next use of \texttt{getgrent} will return the first entry. The return value is unspecified.
getgrent  
Return the next entry in the group database, using the stream set by setgrent.

endgrent  
Closes the stream used by getgrent. The return value is unspecified.

setgr [arg]  
scm_setgrent (arg)  
If called with a true argument, initialize or reset the group data stream. Otherwise, close the stream. The setgrent and endgrent procedures are implemented on top of this.

getgr [group]  
scm_getgrgid (group)  
Look up an entry in the group database. group can be an integer, a string, or omitted, giving the behaviour of getgrgid, getgrnam or getgrent respectively.

In addition to the accessor procedures for the user database, the following shortcut procedure is also available.

getlogin  
scm_getlogin ()  
Return a string containing the name of the user logged in on the controlling terminal of the process, or #f if this information cannot be obtained.

7.2.5 Time

current-time  
scm_current_time ()  
Return the number of seconds since 1970-01-01 00:00:00 UTC, excluding leap seconds.

gmtimeofday  
scm_gettimeofday ()  
Return a pair containing the number of seconds and microseconds since 1970-01-01 00:00:00 UTC, excluding leap seconds. Note: whether true microsecond resolution is available depends on the operating system.

The following procedures either accept an object representing a broken down time and return a selected component, or accept an object representing a broken down time and a value and set the component to the value. The numbers in parentheses give the usual range.

tm:sec tm  
set-tm:sec tm val  
Seconds (0-59).

tm:min tm  
set-tm:min tm val  
Minutes (0-59).

tm:hour tm  
set-tm:hour tm val  
Hours (0-23).
tm:mday tm
set-tm:mday tm val
   Day of the month (1-31).

tm:mon tm
set-tm:mon tm val
   Month (0-11).

tm:year tm
set-tm:year tm val
   Year (70-), the year minus 1900.

tm:wday tm
set-tm:wday tm val
   Day of the week (0-6) with Sunday represented as 0.

tm:yday tm
set-tm:yday tm val
   Day of the year (0-364, 365 in leap years).

tm:isdst tm
set-tm:isdst tm val
   Daylight saving indicator (0 for “no”, greater than 0 for “yes”, less than 0 for “unknown”).

tm:gmtoff tm
set-tm:gmtoff tm val
   Time zone offset in seconds west of UTC (-46800 to 43200). For example on East coast USA (zone ‘EST+5’) this would be 18000 (ie. 5 × 60 × 60) in winter, or 14400 (ie. 4 × 60 × 60) during daylight savings.

Note tm:gmtoff is not the same as tm_gmtoff in the C tm structure. tm_gmtoff is seconds east and hence the negative of the value here.

tm:zone tm
set-tm:zone tm val
   Time zone label (a string), not necessarily unique.

localtime time [zone]
scm_localtime (time, zone)
   Return an object representing the broken down components of time, an integer like the one returned by current-time. The time zone for the calculation is optionally specified by zone (a string), otherwise the TZ environment variable or the system default is used.

gmtime time
scm_gmtime (time)
   Return an object representing the broken down components of time, an integer like the one returned by current-time. The values are calculated for UTC.
**mktime sbd-time [zone]**  
**Scheme Procedure**

**scm_mktime (sbd_time, zone)**  
**C Function**

For a broken down time object `sbd-time`, return a pair the car of which is an integer time like `current-time`, and the cdr of which is a new broken down time with normalized fields.

`zone` is a timezone string, or the default is the `TZ` environment variable or the system default (see Section “Specifying the Time Zone with `TZ`” in GNU C Library Reference Manual). `sbd-time` is taken to be in that `zone`.

The following fields of `sbd-time` are used: `tm:year, tm:mon, tm:mday, tm:hour, tm:min, tm:sec, tm:isdst`. The values can be outside their usual ranges. For example `tm:hour` normally goes up to 23, but a value say 33 would mean 9 the following day.

`tm:isdst` in `sbd-time` says whether the time given is with daylight savings or not. This is ignored if `zone` doesn’t have any daylight savings adjustment amount.

The broken down time in the return normalizes the values of `sbd-time` by bringing them into their usual ranges, and using the actual daylight savings rule for that time in `zone` (which may differ from what `sbd-time` had). The easiest way to think of this is that `sbd-time` plus `zone` converts to the integer UTC time, then a `localtime` is applied to get the normal presentation of that time, in `zone`.

**tzset**  
**Scheme Procedure**

**scm_tzset ()**  
**C Function**

Initialize the timezone from the `TZ` environment variable or the system default. It’s not usually necessary to call this procedure since it’s done automatically by other procedures that depend on the timezone.

**strftime format tm**  
**Scheme Procedure**

**scm_strftime (format, tm)**  
**C Function**

Return a string which is broken-down time structure `tm` formatted according to the given `format` string.

`format` contains field specifications introduced by a `%` character. See Section “Formatting Calendar Time” in The GNU C Library Reference Manual, or `man 3 strftime`, for the available formatting.

\[
(\text{strftime} "\%c" (\text{localtime} (\text{current-time})))
\Rightarrow "\text{Mon Mar 11 20:17:43 2002}"
\]

If `setlocale` has been called (see Section 7.2.13 [Locales], page 542), month and day names are from the current locale and in the locale character set.

**strptime format string**  
**Scheme Procedure**

**scm_strptime (format, string)**  
**C Function**

Performs the reverse action to `strftime`, parsing `string` according to the specification supplied in `format`. The interpretation of month and day names is dependent on the current locale. The value returned is a pair. The CAR has an object with time components in the form returned by `localtime` or `gmtime`, but the time zone components are not usefully set. The CDR reports the number of characters from `string` which were used for the conversion.
internal-time-units-per-second  [Variable]
The value of this variable is the number of time units per second reported by the
following procedures.

times  [Scheme Procedure]
scm_times ()  [C Function]
Return an object with information about real and processor time. The following
procedures accept such an object as an argument and return a selected component:

tms:clock tms  [Scheme Procedure]
The current real time, expressed as time units relative to an arbitrary base.

tms:utime tms  [Scheme Procedure]
The CPU time units used by the calling process.

tms:stime tms  [Scheme Procedure]
The CPU time units used by the system on behalf of the calling process.

tms:cutime tms  [Scheme Procedure]
The CPU time units used by terminated child processes of the calling process,
whose status has been collected (e.g., using waitpid).

tms:cstime tms  [Scheme Procedure]
Similarly, the CPU times units used by the system on behalf of terminated child
processes.

get-internal-real-time  [Scheme Procedure]
scm_get_internal_real_time ()  [C Function]
Return the number of time units since the interpreter was started.

get-internal-run-time  [Scheme Procedure]
scm_get_internal_run_time ()  [C Function]
Return the number of time units of processor time used by the interpreter. Both
system and user time are included but subprocesses are not.

7.2.6 Runtime Environment

program-arguments  [Scheme Procedure]
command-line  [Scheme Procedure]
set-program-arguments  [Scheme Procedure]
scm_program_arguments ()  [C Function]
scm_set_program_arguments_scm (lst)  [C Function]
Get the command line arguments passed to Guile, or set new arguments.
The arguments are a list of strings, the first of which is the invoked program name.
This is just "guile" (or the executable path) when run interactively, or it’s the script
name when running a script with -s (see Section 4.2 [Invoking Guile], page 35).

guile -L /my/extra/dir -s foo.scm abc def

(program-arguments) ⇒ ("foo.scm" "abc" "def")
set-program-arguments allows a library module or similar to modify the arguments, for example to strip options it recognises, leaving the rest for the mainline.

The argument list is held in a fluid, which means it’s separate for each thread. Neither the list nor the strings within it are copied at any point and normally should not be mutated.

The two names program-arguments and command-line are an historical accident, they both do exactly the same thing. The name scm_set_program_arguments_scm has an extra _scm on the end to avoid clashing with the C function below.

void scm_set_program_arguments (int argc, char **argv, char *first)  
[C Function]
Set the list of command line arguments for program-arguments and command-line above.

argv is an array of null-terminated strings, as in a C main function. argc is the number of strings in argv, or if it’s negative then a NULL in argv marks its end.

first is an extra string put at the start of the arguments, or NULL for no such extra. This is a convenient way to pass the program name after advancing argv to strip option arguments. Eg.

```
{  
    char *progname = argv[0];  
    for (argv++; argv[0] != NULL && argv[0][0] == '-' && argv++)  
    {  
        /* munch option ... */  
    }  
    /* remaining args for scheme level use */  
    scm_set_program_arguments (-1, argv, progname);  
}
```

This sort of thing is often done at startup under scm_boot_guile with options handled at the C level removed. The given strings are all copied, so the C data is not accessed again once scm_set_program_arguments returns.

getenv name  
[Scheme Procedure]
scm_getenv (name)  
[C Function]
Looks up the string name in the current environment. The return value is #f unless a string of the form NAME=VALUE is found, in which case the string VALUE is returned.

setenv name value  
[Scheme Procedure]
Modifies the environment of the current process, which is also the default environment inherited by child processes.

If value is #f, then name is removed from the environment. Otherwise, the string name=value is added to the environment, replacing any existing string with name matching name.

The return value is unspecified.

unsetenv name  
[Scheme Procedure]
Remove variable name from the environment. The name can not contain a ‘=’ character.
environ [env]

scm_environ (env)

If env is omitted, return the current environment (in the Unix sense) as a list of strings. Otherwise set the current environment, which is also the default environment for child processes, to the supplied list of strings. Each member of env should be of the form name=value and values of name should not be duplicated. If env is supplied then the return value is unspecified.

putenv str

scm_putenv (str)

Modifies the environment of the current process, which is also the default environment inherited by child processes. If str is of the form NAME=VALUE then it will be written directly into the environment, replacing any existing environment string with name matching NAME. If str does not contain an equal sign, then any existing string with name matching str will be removed.

The return value is unspecified.

7.2.7 Processes

chdir str

scm_chdir (str)

Change the current working directory to str. The return value is unspecified.

getcwd

scm_getcwd ()

Return the name of the current working directory.

umask [mode]

scm_umask (mode)

If mode is omitted, returns a decimal number representing the current file creation mask. Otherwise the file creation mask is set to mode and the previous value is returned. See Section “Assigning File Permissions” in The GNU C Library Reference Manual, for more on how to use umasks.

E.g., (umask #o022) sets the mask to octal 22/decimal 18.

chroot path

scm_chroot (path)

Change the root directory to that specified in path. This directory will be used for path names beginning with /. The root directory is inherited by all children of the current process. Only the superuser may change the root directory.

getpid

scm_getpid ()

Return an integer representing the current process ID.

getgroups

scm_getgroups ()

Return a vector of integers representing the current supplementary group IDs.
getppid
scm_getppid ()
   Return an integer representing the process ID of the parent process.

getuid
scm_getuid ()
   Return an integer representing the current real user ID.

getgid
scm_getgid ()
   Return an integer representing the current real group ID.

geteuid
scm_geteuid ()
   Return an integer representing the current effective user ID. If the system does not
   support effective IDs, then the real ID is returned. (provided? 'EIDs) reports
   whether the system supports effective IDs.

getegid
scm_getegid ()
   Return an integer representing the current effective group ID. If the system does
   not support effective IDs, then the real ID is returned. (provided? 'EIDs) reports
   whether the system supports effective IDs.

setgroups vec
scm_setgroups (vec)
   Set the current set of supplementary group IDs to the integers in the given vector
   vec. The return value is unspecified.

   Generally only the superuser can set the process group IDs (see Section “Setting

setuid id
scm_setuid (id)
   Sets both the real and effective user IDs to the integer id, provided the process has
   appropriate privileges. The return value is unspecified.

setgid id
scm_setgid (id)
   Sets both the real and effective group IDs to the integer id, provided the process has
   appropriate privileges. The return value is unspecified.

seteuid id
scm_seteuid (id)
   Sets the effective user ID to the integer id, provided the process has appropriate
   privileges. If effective IDs are not supported, the real ID is set instead—(provided?
   'EIDs) reports whether the system supports effective IDs. The return value is un-
   specified.
**setegid id**

**scm_setegid (id)**

Sets the effective group ID to the integer id, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—(provided? 'EIDs) reports whether the system supports effective IDs. The return value is unspecified.

**getpgrp**

**scm_getpgrp ()**

Return an integer representing the current process group ID. This is the POSIX definition, not BSD.

**setpgid pid pgid**

**scm_setpgid (pid, pgid)**

Move the process pid into the process group pgid. pid or pgid must be integers: they can be zero to indicate the ID of the current process. Fails on systems that do not support job control. The return value is unspecified.

**setsid**

**scm_setsid ()**

Creates a new session. The current process becomes the session leader and is put in a new process group. The process will be detached from its controlling terminal if it has one. The return value is an integer representing the new process group ID.

**getsid pid**

**scm_getsid (pid)**

Returns the session ID of process pid. (The session ID of a process is the process group ID of its session leader.)

**waitpid pid [options]**

**scm_waitpid (pid, options)**

This procedure collects status information from a child process which has terminated or (optionally) stopped. Normally it will suspend the calling process until this can be done. If more than one child process is eligible then one will be chosen by the operating system.

The value of pid determines the behaviour:

- **pid greater than 0**
  Request status information from the specified child process.

- **pid equal to -1 or WAIT_ANY**
  Request status information for any child process.

- **pid equal to 0 or WAIT_MYPGRP**
  Request status information for any child process in the current process group.

- **pid less than -1**
  Request status information for any child process whose process group ID is −pid.
The \textit{options} argument, if supplied, should be the bitwise OR of the values of zero or more of the following variables:

\begin{itemize}
\item \texttt{WNOHANG} \hspace{1cm} [Variable]
\quad Return immediately even if there are no child processes to be collected.
\item \texttt{WUNTRACED} \hspace{1cm} [Variable]
\quad Report status information for stopped processes as well as terminated processes.
\end{itemize}

The return value is a pair containing:
1. The process ID of the child process, or 0 if \texttt{WNOHANG} was specified and no process was collected.
2. The integer status value (see Section \textquotedblleft Process Completion Status\textquotedblright\ in \textit{The GNU C Library Reference Manual}).

The following three functions can be used to decode the integer status value returned by \texttt{waitpid}.

\begin{description}
\item [status:exit-val \texttt{status}] \hspace{1cm} [Scheme Procedure]
\quad Return the exit status value, as would be set if a process ended normally through a call to \texttt{exit} or \texttt{_exit}, if any, otherwise \#f.
\item [status:term-sig \texttt{status}] \hspace{1cm} [Scheme Procedure]
\quad Return the signal number which terminated the process, if any, otherwise \#f.
\item [status:stop-sig \texttt{status}] \hspace{1cm} [Scheme Procedure]
\quad Return the signal number which stopped the process, if any, otherwise \#f.
\end{description}

\begin{description}
\item [scm\_status\_exit\_val \texttt{(status)}] \hspace{1cm} [C Function]
\item [scm\_status\_term\_sig \texttt{(status)}] \hspace{1cm} [C Function]
\item [scm\_status\_stop\_sig \texttt{(status)}] \hspace{1cm} [C Function]
\end{description}

\begin{description}
\item [system \texttt{[cmd]}] \hspace{1cm} [Scheme Procedure]
\quad Execute \texttt{cmd} using the operating system’s “command processor”. Under Unix this is usually the default shell \texttt{sh}. The value returned is \texttt{cmd}'s exit status as returned by \texttt{waitpid}, which can be interpreted using the functions above.
\quad If \texttt{system} is called without arguments, return a boolean indicating whether the command processor is available.
\item [system* \texttt{arg1 arg2 \ldots}] \hspace{1cm} [Scheme Procedure]
\quad Execute the command indicated by \texttt{arg1 arg2 \ldots}. The first element must be a string indicating the command to be executed, and the remaining items must be strings representing each of the arguments to that command.
\quad This function returns the exit status of the command as provided by \texttt{waitpid}. This value can be handled with \texttt{status:exit-val} and the related functions.
\item [scm\_system\_star \texttt{(args)}] \hspace{1cm} [C Function]
\quad \texttt{system*} is similar to \texttt{system}, but accepts only one string per-argument, and performs no shell interpretation. The command is executed using fork and execlp. Accordingly this function may be safer than \texttt{system} in situations where shell interpretation is not required.
\end{description}

Example: (system* "echo" "foo" "bar")
quit \texttt{[status]} \quad \begin{tabular}{l} (Scheme Procedure) \end{tabular}

\texttt{exit \ [status]} \quad \begin{tabular}{l} (Scheme Procedure) \end{tabular}

Terminate the current process with proper unwinding of the Scheme stack. The exit status zero if \texttt{status} is not supplied. If \texttt{status} is supplied, and it is an integer, that integer is used as the exit status. If \texttt{status} is \#t or \#f, the exit status is \texttt{EXIT\_SUCCESS} or \texttt{EXIT\_FAILURE}, respectively.

The procedure \texttt{exit} is an alias of \texttt{quit}. They have the same functionality.

\texttt{EXIT\_SUCCESS} \quad \begin{tabular}{l} (Scheme Variable) \end{tabular}

\texttt{EXIT\_FAILURE} \quad \begin{tabular}{l} (Scheme Variable) \end{tabular}

These constants represent the standard exit codes for success (zero) or failure (one.)

\texttt{primitive\_exit \ [status]} \quad \begin{tabular}{l} (Scheme Procedure) \end{tabular}

\texttt{primitive\_\_exit \ [status]} \quad \begin{tabular}{l} (Scheme Procedure) \end{tabular}

\texttt{scm\_primitive\_exit \ (status)} \quad \begin{tabular}{l} (C Function) \end{tabular}

\texttt{scm\_primitive\_\_exit \ (status)} \quad \begin{tabular}{l} (C Function) \end{tabular}

Terminate the current process without unwinding the Scheme stack. The exit status is \texttt{status} if supplied, otherwise zero.

\texttt{primitive\_exit} uses the C \texttt{exit} function and hence runs usual C level cleanups (flush output streams, call \texttt{atexit} functions, etc, see Section “Normal Termination” in \textit{The GNU C Library Reference Manual}).

\texttt{primitive\_\_exit} is the \texttt{_exit} system call (see Section “Termination Internals” in \textit{The GNU C Library Reference Manual}). This terminates the program immediately, with neither Scheme-level nor C-level cleanups.

The typical use for \texttt{primitive\_\_exit} is from a child process created with \texttt{primitive\_fork}. For example in a Gdk program the child process inherits the X server connection and a C-level \texttt{atexit} cleanup which will close that connection. But closing in the child would upset the protocol in the parent, so \texttt{primitive\_\_exit} should be used to exit without that.

\texttt{execl} \ \texttt{filename \ arg \ldots} \quad \begin{tabular}{l} (Scheme Procedure) \end{tabular}

\texttt{scm\_execl \ (filename, args)} \quad \begin{tabular}{l} (C Function) \end{tabular}

Executes the file named by \texttt{filename} as a new process image. The remaining arguments are supplied to the process; from a C program they are accessible as the \texttt{argv} argument to \texttt{main}. Conventionally the first \texttt{arg} is the same as \texttt{filename}. All arguments must be strings.

If \texttt{arg} is missing, \texttt{filename} is executed with a null argument list, which may have system-dependent side-effects.

This procedure is currently implemented using the \texttt{execv} system call, but we call it \texttt{execl} because of its Scheme calling interface.

\texttt{execlp} \ \texttt{filename \ arg \ldots} \quad \begin{tabular}{l} (Scheme Procedure) \end{tabular}

\texttt{scm\_execlp \ (filename, args)} \quad \begin{tabular}{l} (C Function) \end{tabular}

Similar to \texttt{execl}, however if \texttt{filename} does not contain a slash then the file to execute will be located by searching the directories listed in the \texttt{PATH} environment variable.

This procedure is currently implemented using the \texttt{execvp} system call, but we call it \texttt{execlp} because of its Scheme calling interface.
execle  filename  env  arg  ...  

scm_execle  (filename, env, args)

Similar to exec1, but the environment of the new process is specified by env, which must be a list of strings as returned by the environ procedure.

This procedure is currently implemented using the execve system call, but we call it execle because of its Scheme calling interface.

primitive-fork

scm_fork()

Creates a new “child” process by duplicating the current “parent” process. In the child the return value is 0. In the parent the return value is the integer process ID of the child.

Note that it is unsafe to fork a process that has multiple threads running, as only the thread that calls primitive-fork will persist in the child. Any resources that other threads held, such as locked mutexes or open file descriptors, are lost. Indeed, POSIX specifies that only async-signal-safe procedures are safe to call after a multithreaded fork, which is a very limited set. Guile issues a warning if it detects a fork from a multi-threaded program.

If you are going to exec soon after forking, the procedures in (ice-9 popen) may be useful to you, as they fork and exec within an async-signal-safe function carefully written to ensure robust program behavior, even in the presence of threads. See Section 7.2.10 [Pipes], page 524, for more.

This procedure has been renamed from fork to avoid a naming conflict with the scsh fork.

nice incr

scm_nice (incr)

Increment the priority of the current process by incr. A higher priority value means that the process runs less often. The return value is unspecified.

setpriority which who prio

scm_setpriority (which, who, prio)

Set the scheduling priority of the process, process group or user, as indicated by which and who. which is one of the variables PRIO_PROCESS, PRIO_PGRP or PRIO_USER, and who is interpreted relative to which (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user identifier for PRIO_USER. A zero value of who denotes the current process, process group, or user. prio is a value in the range $[-20, 20]$. The default priority is 0; lower priorities (in numerical terms) cause more favorable scheduling. Sets the priority of all of the specified processes. Only the super-user may lower priorities. The return value is not specified.

getpriority which who

scm_getpriority (which, who)

Return the scheduling priority of the process, process group or user, as indicated by which and who. which is one of the variables PRIO_PROCESS, PRIO_PGRP or PRIO_USER, and who should be interpreted depending on which (a process identifier for
PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user identifier for PRIO_USER). A zero value of who denotes the current process, process group, or user. Return the highest priority (lowest numerical value) of any of the specified processes.

**getaffinity pid**

getaffinity (pid)

Return a bitvector representing the CPU affinity mask for process *pid*. Each CPU the process has affinity with has its corresponding bit set in the returned bitvector. The number of bits set is a good estimate of how many CPUs Guile can use without stepping on other processes’ toes.

Currently this procedure is only defined on GNU variants (see Section “CPU Affinity” in The GNU C Library Reference Manual).

**setaffinity pid mask**

setaffinity (pid, mask)

Install the CPU affinity mask *mask*, a bitvector, for the process or thread with ID *pid*. The return value is unspecified.

Currently this procedure is only defined on GNU variants (see Section “CPU Affinity” in The GNU C Library Reference Manual).

See Section 6.22.1 [Threads], page 436, for information on how get the number of processors available on a system.

### 7.2.8 Signals

The following procedures raise, handle and wait for signals.

Scheme code signal handlers are run via an async (see Section 6.22.3 [Asyncs], page 439), so they’re called in the handler’s thread at the next safe opportunity. Generally this is after any currently executing primitive procedure finishes (which could be a long time for primitives that wait for an external event).

**kill pid sig**

kill (pid, sig)

Sends a signal to the specified process or group of processes.

*pid* specifies the processes to which the signal is sent:

*pid* greater than 0

The process whose identifier is *pid*.

*pid* equal to 0

All processes in the current process group.

*pid* less than -1

The process group whose identifier is -*pid*.

*pid* equal to -1

If the process is privileged, all processes except for some special system processes. Otherwise, all processes with the current effective user ID.

*sig* should be specified using a variable corresponding to the Unix symbolic name, e.g.,
SIGHUP
Hang-up signal.

SIGINT
Interrupt signal.

A full list of signals on the GNU system may be found in Section “Standard Signals” in *The GNU C Library Reference Manual*.

**raise sig**

**scm_raise (sig)**

Sends a specified signal *sig* to the current process, where *sig* is as described for the *kill* procedure.

**sigaction signum [handler [flags [thread]]]**

**scm_sigaction (signum, handler, flags)**

**scm_sigaction_for_thread (signum, handler, flags, thread)**

Install or report the signal handler for a specified signal.

*signum* is the signal number, which can be specified using the value of variables such as **SIGINT**.

If *handler* is omitted, **sigaction** returns a pair: the CAR is the current signal handler, which will be either an integer with the value **SIG_DFL** (default action) or **SIG_IGN** (ignore), or the Scheme procedure which handles the signal, or **#f** if a non-Scheme procedure handles the signal. The CDR contains the current **sigaction** flags for the handler.

If *handler* is provided, it is installed as the new handler for *signum*. *handler* can be a Scheme procedure taking one argument, or the value of **SIG_DFL** (default action) or **SIG_IGN** (ignore), or **#f** to restore whatever signal handler was installed before **sigaction** was first used. When a scheme procedure has been specified, that procedure will run in the given *thread*. When no thread has been given, the thread that made this call to **sigaction** is used.

*flags* is a **logior** (see Section 6.6.2.13 [Bitwise Operations], page 125) of the following (where provided by the system), or 0 for none.

**SA_NOCLDSTOP**

By default, **SIGCHLD** is signalled when a child process stops (ie. receives **SIGSTOP**), and when a child process terminates. With the **SA_NOCLDSTOP** flag, **SIGCHLD** is only signalled for termination, not stopping.

**SA_NOCLDSTOP** has no effect on signals other than **SIGCHLD**.

**SA_RESTART**

If a signal occurs while in a system call, deliver the signal then restart the system call (as opposed to returning an **EINTR** error from that call).

Guile handles signals asynchronously. When it receives a signal, the synchronous signal handler just records the fact that a signal was received and sets a flag to tell the relevant Guile thread that it has a pending signal. When the Guile thread checks
the pending-interrupt flag, it will arrange to run the asynchronous part of the signal handler, which is the handler attached by `sigaction`.

This strategy has some perhaps-unexpected interactions with the `SA_RESTART` flag, though: because the synchronous handler doesn’t do very much, and notably it doesn’t run the Guile handler, it’s impossible to interrupt a thread stuck in a long-running system call via a signal handler that is installed with `SA_RESTART`: the synchronous handler just records the pending interrupt, but then the system call resumes and Guile doesn’t have a chance to actually check the flag and run the asynchronous handler. That’s just how it is.

The return value is a pair with information about the old handler as described above. This interface does not provide access to the “signal blocking” facility. Maybe this is not needed, since the thread support may provide solutions to the problem of consistent access to data structures.

```
restore-signals                           [Scheme Procedure]
scm_restore_signals ()                  [C Function]
  Return all signal handlers to the values they had before any call to `sigaction` was made. The return value is unspecified.

alarm i                                  [Scheme Procedure]
scm_alarm (i)                            [C Function]
  Set a timer to raise a SIGALRM signal after the specified number of seconds (an integer). It’s advisable to install a signal handler for SIGALRM beforehand, since the default action is to terminate the process.
  The return value indicates the time remaining for the previous alarm, if any. The new value replaces the previous alarm. If there was no previous alarm, the return value is zero.

pause                                    [Scheme Procedure]
scm_pause ()                             [C Function]
  Pause the current process (thread?) until a signal arrives whose action is to either terminate the current process or invoke a handler procedure. The return value is unspecified.

sleep secs                               [Scheme Procedure]
sleep secs                               [C Function]
  Wait the given period secs seconds or usecs microseconds (both integers). If a signal arrives the wait stops and the return value is the time remaining, in seconds or microseconds respectively. If the period elapses with no signal the return is zero.
  On most systems the process scheduler is not microsecond accurate and the actual period slept by `usleep` might be rounded to a system clock tick boundary, which might be 10 milliseconds for instance.

  See `scm_std_sleep` and `scm_std_usleep` for equivalents at the C level (see Section 6.22.6 [Blocking], page 446).
```
getitimer which_timer
setitimer which_timer interval_seconds
    interval_microseconds value_seconds value_microseconds

scm_getitimer (which_timer)
scm_setitimer (which_timer, interval_seconds,
    interval_microseconds, value_seconds, value_microseconds)

Get or set the periods programmed in certain system timers. These timers have two settings. The first setting, the interval, is the value at which the timer will be reset when the current timer expires. The second is the current value of the timer, indicating when the next expiry will be signalled.

which_timer is one of the following values:

ITIMER_REAL
    A real-time timer, counting down elapsed real time. At zero it raises SIGALRM. This is like alarm above, but with a higher resolution period.

ITIMER_VIRTUAL
    A virtual-time timer, counting down while the current process is actually using CPU. At zero it raises SIGVTALRM.

ITIMER_PROF
    A profiling timer, counting down while the process is running (like ITIMER_VIRTUAL) and also while system calls are running on the process’s behalf. At zero it raises a SIGPROF.
    This timer is intended for profiling where a program is spending its time (by looking where it is when the timer goes off).

getitimer returns the restart timer value and its current value, as a list containing two pairs. Each pair is a time in seconds and microseconds: ((interval_secs . interval_usecs) (value_secs . value_usecs)).

setitimer sets the timer values similarly, in seconds and microseconds (which must be integers). The interval value can be zero to have the timer run down just once. The return value is the timer’s previous setting, in the same form as getitimer returns.

(setitimer ITIMER_REAL
    5 500000 ;; Raise SIGALRM every 5.5 seconds
    2 0) ;; with the first SIGALRM in 2 seconds

Although the timers are programmed in microseconds, the actual accuracy might not be that high.

Note that ITIMER_PROF and ITIMER_VIRTUAL are not functional on all platforms and may always error when called. (provided? 'ITIMER_PROF) and (provided? 'ITIMER_VIRTUAL) can be used to test if those timers are supported on the given host. ITIMER_REAL is supported on all platforms that support setitimer.

7.2.9 Terminals and Ptys

isatty? port
scm_isatty_p (port)

Return #t if port is using a serial non–file device, otherwise #f.
ttynamem port

scm_ttynamem (port)

Return a string with the name of the serial terminal device underlying port.

ctermid

scm_ctermid ()

Return a string containing the file name of the controlling terminal for the current process.

tcgetpgrp port

scm_tcgetpgrp (port)

Return the process group ID of the foreground process group associated with the terminal open on the file descriptor underlying port.

If there is no foreground process group, the return value is a number greater than 1 that does not match the process group ID of any existing process group. This can happen if all of the processes in the job that was formerly the foreground job have terminated, and no other job has yet been moved into the foreground.

tcsetpgrp port pgid

scm_tcsetpgrp (port, pgid)

Set the foreground process group ID for the terminal used by the file descriptor underlying port to the integer pgid. The calling process must be a member of the same session as pgid and must have the same controlling terminal. The return value is unspecified.

7.2.10 Pipes

The following procedures are similar to the popen and pclose system routines. The code is in a separate “popen” module:

(use-modules (ice-9 popen))

open-pipe command mode

open-pipe* mode prog [args...]

Execute a command in a subprocess, with a pipe to it or from it, or with pipes in both directions.

open-pipe runs the shell command using ‘/bin/sh -c’. open-pipe* executes prog directly, with the optional args arguments (all strings).

mode should be one of the following values. OPEN_READ is an input pipe, i.e. to read from the subprocess. OPEN_WRITE is an output pipe, i.e. to write to it.

OPEN_READ

OPEN_WRITE

OPEN_BOTH

For an input pipe, the child’s standard output is the pipe and standard input is inherited from current-input-port. For an output pipe, the child’s standard input is the pipe and standard output is inherited from current-output-port. In all cases

1 This module is only available on systems where the popen feature is provided (see Section 6.23.2.2 [Common Feature Symbols], page 452).
the child’s standard error is inherited from \texttt{current-error-port} (see Section 6.14.9 [Default Ports], page 340).

If those \texttt{current-X-ports} are not files of some kind, and hence don’t have file descriptors for the child, then \texttt{/dev/null} is used instead.

Care should be taken with \texttt{OPEN_BOTH}, a deadlock will occur if both parent and child are writing, and waiting until the write completes before doing any reading. Each direction has \texttt{PIPE_BUF} bytes of buffering (see Section 6.14.6 [Buffering], page 336), which will be enough for small writes, but not for say putting a big file through a filter.

\textbf{open-input-pipe \textit{command}}

Equivalent to \texttt{open-pipe} with mode \texttt{OPEN_READ}.

\begin{verbatim}
(let* ((port (open-input-pipe "date --utc"))
       (str (read-line port)))
  (close-pipe port)
  str)
⇒ "Mon Mar 11 20:10:44 UTC 2002"
\end{verbatim}

\textbf{open-output-pipe \textit{command}}

Equivalent to \texttt{open-pipe} with mode \texttt{OPEN_WRITE}.

\begin{verbatim}
(let ((port (open-output-pipe "lpr")))
  (display "Something for the line printer.\n" port)
  (if (not (eqv? 0 (status:exit-val (close-pipe port))))
      (error "Cannot print")))
\end{verbatim}

\textbf{open-input-output-pipe \textit{command}}

Equivalent to \texttt{open-pipe} with mode \texttt{OPEN_BOTH}.

\textbf{close-pipe \textit{port}}

Close a pipe created by \texttt{open-pipe}, wait for the process to terminate, and return the wait status code. The status is as per \texttt{waitpid} and can be decoded with \texttt{status:exit-val} etc (see Section 7.2.7 [Processes], page 514).

\texttt{waitpid \texttt{WAIT\_ANY}} should not be used when pipes are open, since it can reap a pipe’s child process, causing an error from a subsequent \texttt{close-pipe}.

\texttt{close-port} (see Section 6.14.1 [Ports], page 328) can close a pipe, but it doesn’t reap the child process.

The garbage collector will close a pipe no longer in use, and reap the child process with \texttt{waitpid}. If the child hasn’t yet terminated the garbage collector doesn’t block, but instead checks again in the next GC.

Many systems have per-user and system-wide limits on the number of processes, and a system-wide limit on the number of pipes, so pipes should be closed explicitly when no longer needed, rather than letting the garbage collector pick them up at some later time.

\textbf{7.2.11 Networking}
7.2.11.1 Network Address Conversion

This section describes procedures which convert internet addresses between numeric and string formats.

IPv4 Address Conversion

An IPv4 Internet address is a 4-byte value, represented in Guile as an integer in host byte order, so that say “0.0.0.1” is 1, or “1.0.0.0” is 16777216.

Some underlying C functions use network byte order for addresses, Guile converts as necessary so that at the Scheme level its host byte order everywhere.

INADDR_ANY [Variable]
For a server, this can be used with bind (see Section 7.2.11.4 [Network Sockets and Communication], page 535) to allow connections from any interface on the machine.

INADDR_BROADCAST [Variable]
The broadcast address on the local network.

INADDR_LOOPBACK [Variable]
The address of the local host using the loopback device, i.e. ‘127.0.0.1’.

inet-netof address [Scheme Procedure]
scm_inet_netof (address) [C Function]
Return the network number part of the given IPv4 Internet address. E.g.,

\[(\text{inet-netof 2130706433}) \Rightarrow 127\]

inet-lnaof address [Scheme Procedure]
scm_lnaof (address) [C Function]
Return the local-address-with-network part of the given IPv4 Internet address, using the obsolete class A/B/C system. E.g.,

\[(\text{inet-lnaof 2130706433}) \Rightarrow 1\]

inet-makeaddr net lna [Scheme Procedure]
scm_inet_makeaddr (net, lna) [C Function]
Make an IPv4 Internet address by combining the network number net with the local-address-within-network number lna. E.g.,

\[(\text{inet-makeaddr 127 1}) \Rightarrow 2130706433\]

IPv6 Address Conversion

An IPv6 Internet address is a 16-byte value, represented in Guile as an integer in host byte order, so that say “::1” is 1.

inet-ntop family address [Scheme Procedure]
scm_inet_ntop (family, address) [C Function]
Convert a network address from an integer to a printable string. family can be AF_INET or AF_INET6. E.g.,

\[(\text{inet-ntop AF_INET 2130706433}) \Rightarrow "127.0.0.1"\]
\[(\text{inet-ntop AF_INET6 (- (expt 2 128) 1)})\]
**inet-pton** *family address*  
*[Scheme Procedure]*

**scm_inet-pton** *(family, address)*  
*[C Function]*

Convert a string containing a printable network address to an integer address. *family* can be **AF_INET** or **AF_INET6**. E.g.,

\[
\begin{align*}
\text{(inet-pton AF_INET "127.0.0.1") } & \Rightarrow 2130706433 \\
\text{(inet-pton AF_INET6 "::1") } & \Rightarrow 1
\end{align*}
\]

### 7.2.11.2 Network Databases

This section describes procedures which query various network databases. Care should be taken when using the database routines since they are not reentrant.

**getaddrinfo**

The **getaddrinfo** procedure maps host and service names to socket addresses and associated information in a protocol-independent way.

**getaddrinfo** *name service [hint_flags [hint_family  
[hint_socktype [hint_protocol]]]]*  
*[Scheme Procedure]*

**scm_getaddrinfo** *(name, service, hint_flags, hint_family, 
hint_socktype, hint_protocol)*  
*[C Function]*

Return a list of **addrinfo** structures containing a socket address and associated information for host *name* and/or *service* to be used in creating a socket with which to address the specified service.

\[
\begin{align*}
\text{(let* ((ai (car (getaddrinfo "www.gnu.org" "http")))))} \\
\text{ (s (socket (addrinfo:fam ai) (addrinfo:socktype ai) (addrinfo:protocol ai)))} \\
\text{ (connect s (addrinfo:addr ai))} \\
\text{ s)}
\end{align*}
\]

When *service* is omitted or is **#f**, return network-level addresses for *name*. When *name* is **#f** *service* must be provided and service locations local to the caller are returned.

Additional hints can be provided. When specified, *hint_flags* should be a bitwise-or of zero or more constants among the following:

**AI_PASSIVE**  
Socket address is intended for **bind**.

**AI_CANONNAME**  
Request for canonical host name, available via **addrinfo:canonname**. This makes sense mainly when DNS lookups are involved.

**AI_NUMERICHOST**  
Specifies that *name* is a numeric host address string (e.g., "127.0.0.1"), meaning that name resolution will not be used.

**AI_NUMERICSERV**  
Likewise, specifies that *service* is a numeric port string (e.g., "80").

**AI_ADDRCONFIG**  
Return only addresses configured on the local system. It is highly recommended to provide this flag when the returned socket addresses are to
be used to make connections; otherwise, some of the returned addresses could be unreachable or use a protocol that is not supported.

**AI_V4MAPPED**

When looking up IPv6 addresses, return mapped IPv4 addresses if there is no IPv6 address available at all.

**AI_ALL**

If this flag is set along with **AI_V4MAPPED** when looking up IPv6 addresses, return all IPv6 addresses as well as all IPv4 addresses, the latter mapped to IPv6 format.

When given, hint\_family should specify the requested address family, e.g., AF_INET6. Similarly, hint\_socktype should specify the requested socket type (e.g., SOCK_DGRAM), and hint\_protocol should specify the requested protocol (its value is interpreted as in calls to socket).

On error, an exception with key getaddrinfo\_error is thrown, with an error code (an integer) as its argument:

```
(catch 'getaddrinfo-error
  (lambda ()
    (getaddrinfo "www.gnu.org" "gopher"))
  (lambda (key errcode)
    (cond ((= errcode EAI_SERVICE)
        (display "doesn't know about Gopher!\n"))
        ((= errcode EAI_NONAME)
        (display "www.gnu.org not found\n"))
        (else
        (format #t "something wrong: ~a
          (gai-strerror errcode)))))
```

Error codes are:

**EAI_AGAIN**

The name or service could not be resolved at this time. Future attempts may succeed.

**EAI_BADFLAGS**

hint\_flags contains an invalid value.

**EAI_FAIL**

A non-recoverable error occurred when attempting to resolve the name.

**EAI_FAMILY**

hint\_family was not recognized.

**EAI_NONAME**

Either name does not resolve for the supplied parameters, or neither name nor service were supplied.

**EAI_NODATA**

This non-POSIX error code can be returned on some systems (GNU and Darwin, at least), for example when name is known but requests that were made turned out no data. Error handling code should be prepared to handle it when it is defined.
EAI_SERVICE
service was not recognized for the specified socket type.

EAI_SOCKTYPE
hint_socktype was not recognized.

EAI_SYSTEM
A system error occurred. In C, the error code can be found in errno; this value is not accessible from Scheme, but in practice it provides little information about the actual error cause.

Users are encouraged to read the "POSIX specification (http://www.opengroup.org/onlinepubs/9699919799/functions/getaddrinfo.html) for more details.

The following procedures take an addrinfo object as returned by getaddrinfo:

addrinfo:flags ai
Return flags for ai as a bitwise or of AI_ values (see above).

addrinfo:fam ai
Return the address family of ai (a AF_ value).

addrinfo:socktype ai
Return the socket type for ai (a SOCK_ value).

addrinfo:protocol ai
Return the protocol of ai.

addrinfo:addr ai
Return the socket address associated with ai as a sockaddr object (see Section 7.2.11.3 [Network Socket Address], page 533).

addrinfo:canonname ai
Return a string for the canonical name associated with ai if the AI_CANONNAME flag was supplied.

The Host Database
A host object is a structure that represents what is known about a network host, and is the usual way of representing a system’s network identity inside software.

The following functions accept a host object and return a selected component:

hostent:name host
The “official” hostname for host.

hostent:aliases host
A list of aliases for host.

hostent:addrtype host
The host address type, one of the AF constants, such as AF_INET or AF_INET6.

hostent:length host
The length of each address for host, in bytes.
The list of network addresses associated with `host`. For AF_INET these are integer IPv4 address (see Section 7.2.11.1 [Network Address Conversion], page 526).

The following procedures can be used to search the host database. However, `getaddrinfo` should be preferred over them since it’s more generic and thread-safe.

- `gethost [host]` [Scheme Procedure]
- `gethostbyname hostname` [Scheme Procedure]
- `gethostbyaddr address` [Scheme Procedure]
- `scm_gethost (host)` [C Function]

Look up a host by name or address, returning a host object. The `gethost` procedure will accept either a string name or an integer address; if given no arguments, it behaves like `gethostent` (see below). If a name or address is supplied but the address can not be found, an error will be thrown to one of the keys: `host-not-found`, `try-again`, `no-recovery` or `no-data`, corresponding to the equivalent `h_error` values. Unusual conditions may result in errors thrown to the `system-error` or `misc_error` keys.

```
(gethost "www.gnu.org")
⇒ #("www.gnu.org" () 2 4 (3353880842))
```

```
(gethostbyname "www.emacs.org")
⇒ #("emacs.org" ("www.emacs.org") 2 4 (1073448978))
```

The following procedures may be used to step through the host database from beginning to end.

- `sethostent [stayopen]` [Scheme Procedure]
- `gethostent` [Scheme Procedure]
- `endhostent` [Scheme Procedure]
- `sethost [stayopen]` [Scheme Procedure]
- `scm_sethost (stayopen)` [C Function]

The Network Database

The following functions accept an object representing a network and return a selected component:
The following procedures are used to search the network database:

- `getnet [net]` [Scheme Procedure]
- `getnetbyname net-name` [Scheme Procedure]
- `getnetbyaddr net-number` [Scheme Procedure]
- `scm_getnet (net)` [C Function]

Look up a network by name or net number in the network database. The `net-name` argument must be a string, and the `net-number` argument must be an integer. `getnet` will accept either type of argument, behaving like `getnetent` (see below) if no arguments are given.

The following procedures may be used to step through the network database from beginning to end.

- `setnetent [stayopen]` [Scheme Procedure]
  Initialize an internal stream from which network objects may be read. This procedure must be called before any calls to `getnetent`, and may also be called afterward to reset the net entry stream. If `stayopen` is supplied and is not `#f`, the database is not closed by subsequent `getnetbyname` or `getnetbyaddr` calls, possibly giving an efficiency gain.

- `getnetent` [Scheme Procedure]
  Return the next entry from the network database.

- `endnetent` [Scheme Procedure]
  Close the stream used by `getnetent`. The return value is unspecified.

- `setnet [stayopen]` [Scheme Procedure]
- `scm_setnet (stayopen)` [C Function]
  If `stayopen` is omitted, this is equivalent to `endnetent`. Otherwise it is equivalent to `setnetent stayopen`.

The Protocol Database

The following functions accept an object representing a protocol and return a selected component:

- `protoent:name protocol` [Scheme Procedure]
  The “official” protocol name.
protoent:aliases protocol  [Scheme Procedure]
   A list of aliases for the protocol.

protoent:proto protocol  [Scheme Procedure]
   The protocol number.

   The following procedures are used to search the protocol database:

getproto [protocol]  [Scheme Procedure]
getprotobynumber number  [Scheme Procedure]
getprotobynumber name  [Scheme Procedure]
scm_getproto (protocol)  [C Function]
   Look up a network protocol by name or by number. getprotobynumber takes a string
   argument, and getprotobynumber takes an integer argument. getproto will accept
   either type, behaving like getprotoent (see below) if no arguments are supplied.

   The following procedures may be used to step through the protocol database from
   beginning to end.

setprotoent [stayopen]  [Scheme Procedure]
   Initialize an internal stream from which protocol objects may be read. This procedure
   must be called before any calls to getprotoent, and may also be called afterward to
   reset the protocol entry stream. If stayopen is supplied and is not #f, the database
   is not closed by subsequent getprotobyname or getprotobynumber calls, possibly
   giving an efficiency gain.

getprotoent
   Return the next entry from the protocol database.

derprotoent  [Scheme Procedure]
   Close the stream used by getprotoent. The return value is unspecified.

setproto [stayopen]  [Scheme Procedure]
scm_setproto (stayopen)  [C Function]
   If stayopen is omitted, this is equivalent to endprotoent. Otherwise it is equivalent
to setprotoent stayopen.

The Service Database
The following functions accept an object representing a service and return a selected com-
ponent:

servent:name serv  [Scheme Procedure]
   The “official” name of the network service.

servent:aliases serv  [Scheme Procedure]
   A list of aliases for the network service.

servent:port serv  [Scheme Procedure]
   The Internet port used by the service.
servent:proto serv

The protocol used by the service. A service may be listed many times in the database under different protocol names.

The following procedures are used to search the service database:

getserv [name [protocol]]
getservbyname name protocol
getservbyport port protocol
scm_getserv (name, protocol)

Look up a network service by name or by service number, and return a network service object. The protocol argument specifies the name of the desired protocol; if the protocol found in the network service database does not match this name, a system error is signalled.

The getserv procedure will take either a service name or number as its first argument; if given no arguments, it behaves like getservent (see below).

(getserv "imap" "tcp")
⇒ #("imap2" "imap")

(getservbyport 88 "udp")
⇒ #("kerberos" "kerberos5" "krb5")

The following procedures may be used to step through the service database from beginning to end.

setservent [stayopen]

Initialize an internal stream from which service objects may be read. This procedure must be called before any calls to getservent, and may also be called afterward to reset the service entry stream. If stayopen is supplied and is not #f, the database is not closed by subsequent getservbyname or getservbyport calls, possibly giving an efficiency gain.

getservent

Return the next entry from the services database.

dendservent

Close the stream used by getservent. The return value is unspecified.

setserv [stayopen]

A socket address object identifies a socket endpoint for communication. In the case of AF_INET for instance, the socket address object comprises the host address (or interface on the host) and a port number which specifies a particular open socket in a running client or server process. A socket address object can be created with,
make-socket-address  AF_INET  ipv4addr  port  [Scheme Procedure]
make-socket-address  AF_INET6  ipv6addr  port  [flowinfo  scopeid]  [Scheme Procedure]
make-socket-address  AF_UNIX  path  [Scheme Procedure]

scm_make_socket_address  (family, address, arglist)  [C Function]

Return a new socket address object. The first argument is the address family, one of the AF constants, then the arguments vary according to the family.

For AF_INET the arguments are an IPv4 network address number (see Section 7.2.11.1 [Network Address Conversion], page 526), and a port number.

For AF_INET6 the arguments are an IPv6 network address number and a port number. Optional flowinfo and scopeid arguments may be given (both integers, default 0).

For AF_UNIX the argument is a filename (a string).

The C function scm_make_socket_address takes the family and address arguments directly, then arglist is a list of further arguments, being the port for IPv4, port and optional flowinfo and scopeid for IPv6, or the empty list SCM_EOL for Unix domain.

The following functions access the fields of a socket address object,

sockaddr:fam  sa  [Scheme Procedure]
Return the address family from socket address object sa. This is one of the AF constants (e.g. AF_INET).

sockaddr:path  sa  [Scheme Procedure]
For an AF_UNIX socket address object sa, return the filename.

sockaddr:addr  sa  [Scheme Procedure]
For an AF_INET or AF_INET6 socket address object sa, return the network address number.

sockaddr:port  sa  [Scheme Procedure]
For an AF_INET or AF_INET6 socket address object sa, return the port number.

sockaddr:flowinfo  sa  [Scheme Procedure]
For an AF_INET6 socket address object sa, return the flowinfo value.

sockaddr:scopeid  sa  [Scheme Procedure]
For an AF_INET6 socket address object sa, return the scope ID value.

The functions below convert to and from the C struct sockaddr (see Section “Address Formats” in The GNU C Library Reference Manual). That structure is a generic type, an application can cast to or from struct sockaddr_in, struct sockaddr_in6 or struct sockaddr_un according to the address family.

In a struct sockaddr taken or returned, the byte ordering in the fields follows the C conventions (see Section “Byte Order Conversion” in The GNU C Library Reference Manual). This means network byte order for AF_INET host address (sin_addr.s_addr) and port number (sin_port), and AF_INET6 port number (sin6_port). But at the Scheme level these values are taken or returned in host byte order, so the port is an ordinary integer, and the host address likewise is an ordinary integer (as described in Section 7.2.11.1 [Network Address Conversion], page 526).
struct sockaddr * scm_c_make_socket_address (SCM family, SCM address, SCM args, size_t *outsize) [C Function]

Return a newly-malloced struct sockaddr created from arguments like those taken by scm_make_socket_address above.

The size (in bytes) of the struct sockaddr return is stored into *outsize. An application must call free to release the returned structure when no longer required.

SCM scm_from_sockaddr (const struct sockaddr *address, unsigned address_size) [C Function]

Return a Scheme socket address object from the C address structure. address_size is the size in bytes of address.

struct sockaddr * scm_to_sockaddr (SCM address, size_t *address_size) [C Function]

Return a newly-malloced struct sockaddr from a Scheme level socket address object.

The size (in bytes) of the struct sockaddr return is stored into *outsize. An application must call free to release the returned structure when no longer required.

7.2.11.4 Network Sockets and Communication

Socket ports can be created using socket and socketpair. The ports are initially unbuffered, to make reading and writing to the same port more reliable. A buffer can be added to the port using setvbuf (see Section 6.14.6 [Buffering], page 336).

Most systems have limits on how many files and sockets can be open, so it’s strongly recommended that socket ports be closed explicitly when no longer required (see Section 6.14.1 [Ports], page 328).

Some of the underlying C functions take values in network byte order, but the convention in Guile is that at the Scheme level everything is ordinary host byte order and conversions are made automatically where necessary.

socket family style proto [Scheme Procedure]

scm_socket (family, style, proto) [C Function]

Return a new socket port of the type specified by family, style and proto. All three parameters are integers. The possible values for family are as follows, where supported by the system,

PF_UNIX [Variable]
PF_INET [Variable]
PF_INET6 [Variable]

The possible values for style are as follows, again where supported by the system,

SOCK_STREAM [Variable]
SOCK_DGRAM [Variable]
SOCK_RAW [Variable]
SOCK_RDM [Variable]
SOCK_SEQPACKET [Variable]

proto can be obtained from a protocol name using getprotobyname (see Section 7.2.11.2 [Network Databases], page 527). A value of zero means the default protocol, which is usually right.
A socket cannot be used for communication until it has been connected somewhere, usually with either connect or accept below.

**socketpair** family style proto  
[Scheme Procedure]

**scm_socketpair** (family, style, proto)  
[C Function]

Return a pair, the car and cdr of which are two unnamed socket ports connected to each other. The connection is full-duplex, so data can be transferred in either direction between the two.

family, style and proto are as per socket above. But many systems only support socket pairs in the PF_UNIX family. Zero is likely to be the only meaningful value for proto.

**getsockopt** sock level optname  
[Scheme Procedure]

**setsockopt** sock level optname value  
[Scheme Procedure]

**scm_getsockopt** (sock, level, optname)  
[C Function]

**scm_setsockopt** (sock, level, optname, value)  
[C Function]

Get or set an option on socket port sock. getsockopt returns the current value. setsockopt sets a value and the return is unspecified.

level is an integer specifying a protocol layer, either SOL_SOCKET for socket level options, or a protocol number from the IPPROTO constants or getprotoent (see Section 7.2.11.2 [Network Databases], page 527).

**SOL_SOCKET**  
[Variable]

**IPPROTO_IP**  
[Variable]

**IPPROTO_TCP**  
[Variable]

**IPPROTO_UDP**  
[Variable]

optname is an integer specifying an option within the protocol layer.

For SOL_SOCKET level the following optnames are defined (when provided by the system). For their meaning see Section “Socket-Level Options” in The GNU C Library Reference Manual, or man 7 socket.

**SO_DEBUG**  
[Variable]

**SO_REUSEADDR**  
[Variable]

**SO_STYLE**  
[Variable]

**SO_TYPE**  
[Variable]

**SO_ERROR**  
[Variable]

**SO_DONTROUTE**  
[Variable]

**SO_BROADCAST**  
[Variable]

**SO_SNDBUF**  
[Variable]

**SO_RCVBUF**  
[Variable]

**SO_KEEPALIVE**  
[Variable]

**SO_OOBINLINE**  
[Variable]

**SO_NO_CHECK**  
[Variable]

**SO_PRIORITY**  
[Variable]

**SO_REUSEPORT**  
[Variable]

The value taken or returned is an integer.
Chapter 7: Guile Modules

SO_LINGER

[Variable]
The value taken or returned is a pair of integers (ENABLE . TIMEOUT). On old systems without timeout support (ie. without struct linger), only ENABLE has an effect but the value in Guile is always a pair.

For IP level (IPPROTO_IP) the following optnames are defined (when provided by the system). See man ip for what they mean.

IP_MULTICAST_IF

[Variable]
This sets the source interface used by multicast traffic.

IP_MULTICAST_TTL

[Variable]
This sets the default TTL for multicast traffic. This defaults to 1 and should be increased to allow traffic to pass beyond the local network.

IP_ADD_MEMBERSHIP

[Variable]
IP_DROP_MEMBERSHIP

[Variable]
These can be used only with setsockopt, not getsockopt. value is a pair (MULTIADDR . INTERFACEADDR) of integer IPv4 addresses (see Section 7.2.11.1 [Network Address Conversion], page 526). MULTIADDR is a multicast address to be added to or dropped from the interface INTERFACEADDR. INTERFACEADDR can be INADDR_ANY to have the system select the interface. INTERFACEADDR can also be an interface index number, on systems supporting that.

For IPPROTO_TCP level the following optnames are defined (when provided by the system). For their meaning see man 7 tcp.

TCP_NODELAY

[Variable]
TCP_CORK

[Variable]
The value taken or returned is an integer.

shutdown sock how

[Scheme Procedure]
sbcm_shutdown (sock, how)

[C Function]
Sockets can be closed simply by using close-port. The shutdown procedure allows reception or transmission on a connection to be shut down individually, according to the parameter how:

0 Stop receiving data for this socket. If further data arrives, reject it.

1 Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don’t retransmit it if it is lost.

2 Stop both reception and transmission.

The return value is unspecified.

connect sock sockaddr

[Scheme Procedure]
connect sock AF_INET ipv4addr port

[Scheme Procedure]
connect sock AF_INET6 ipv6addr port [flowinfo [scopeid]]

[Scheme Procedure]
connect sock AF_UNIX path

[Scheme Procedure]
scm_connect (sock, fam, address, args)  
[C Function]
Initiate a connection on socket port sock to a given address. The destination is either a socket address object, or arguments the same as make-socket-address would take to make such an object (see Section 7.2.11.3 [Network Socket Address], page 533). Return true unless the socket was configured as non-blocking and the connection could not be made immediately.

(connect sock AF_INET INADDR_LOOPBACK 23)
(connect sock (make-socket-address AF_INET INADDR_LOOPBACK 23))

bind sock sockaddr  
[Scheme Procedure]
bind sock AF_INET ipv4addr port  
[Scheme Procedure]
bind sock AF_INET6 ipv6addr port [flowinfo [scopeid]]  
[Scheme Procedure]
bind sock AF_UNIX path  
[Scheme Procedure]
scm_bind (sock, fam, address, args)  
[C Function]
Bind socket port sock to the given address. The address is either a socket address object, or arguments the same as make-socket-address would take to make such an object (see Section 7.2.11.3 [Network Socket Address], page 533). The return value is unspecified.

Generally a socket is only explicitly bound to a particular address when making a server, i.e. to listen on a particular port. For an outgoing connection the system will assign a local address automatically, if not already bound.

(bind sock AF_INET INADDR_ANY 12345)
(bind sock (make-socket-address AF_INET INADDR_ANY 12345))

listen sock backlog  
[Scheme Procedure]
scm_listen (sock, backlog)  
[C Function]
Enable sock to accept connection requests. backlog is an integer specifying the maximum length of the queue for pending connections. If the queue fills, new clients will fail to connect until the server calls accept to accept a connection from the queue.

The return value is unspecified.

accept sock [flags]  
[Scheme Procedure]
scm_accept (sock)  
[C Function]
Accept a connection from socket port sock which has been enabled for listening with listen above.

If there are no incoming connections in the queue, there are two possible behaviors, depending on whether sock has been configured for non-blocking operation or not:

- If there is no connection waiting and the socket was set to non-blocking mode with the O_NONBLOCK port option (see Section 7.2.2 [Ports and File Descriptors], page 493), return #f directly.
- Otherwise wait until a connection is available.

The return value is a pair. The car is a new socket port, connected and ready to communicate. The cdr is a socket address object (see Section 7.2.11.3 [Network Socket Address], page 533) which is where the remote connection is from (like getpeername below).
flags, if given, may include SOCK_CLOEXEC or SOCK_NONBLOCK, which like O_CLOEXEC and O_NONBLOCK apply to the newly accepted socket.

All communication takes place using the new socket returned. The given sock remains bound and listening, and accept may be called on it again to get another incoming connection when desired.

getsockname sock

scm_getsockname (sock)

Return a socket address object which is the where sock is bound locally. sock may have obtained its local address from bind (above), or if a connect is done with an otherwise unbound socket (which is usual) then the system will have assigned an address.

Note that on many systems the address of a socket in the AF_UNIX namespace cannot be read.

getpeername sock

scm_getpeername (sock)

Return a socket address object which is where sock is connected to, i.e. the remote endpoint.

Note that on many systems the address of a socket in the AF_UNIX namespace cannot be read.

recv! sock buf [flags]

scm_recv (sock, buf, flags)

Receive data from a socket port. sock must already be bound to the address from which data is to be received. buf is a bytevector into which the data will be written. The size of buf limits the amount of data which can be received: in the case of packet protocols, if a packet larger than this limit is encountered then some data will be irrevocably lost.

The optional flags argument is a value or bitwise OR of MSG_OOB, MSG_PEEK, MSG_DONTROUTE etc.

The value returned is the number of bytes read from the socket.

Note that the data is read directly from the socket file descriptor: any unread buffered port data is ignored.

send sock message [flags]

scm_send (sock, message, flags)

Transmit bytevector message on socket port sock. sock must already be bound to a destination address. The value returned is the number of bytes transmitted—it's possible for this to be less than the length of message if the socket is set to be non-blocking. The optional flags argument is a value or bitwise OR of MSG_OOB, MSG_PEEK, MSG_DONTROUTE etc.

Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.
recvfrom! sock buf [flags [start [end]]]  
scm_recvfrom (sock, buf, flags, start, end)  
Receive data from socket port `sock`, returning the originating address as well as the data. This function is usually for datagram sockets, but can be used on stream-oriented sockets too.

The data received is stored in bytevector `buf`, using either the whole bytevector or just the region between the optional `start` and `end` positions. The size of `buf` limits the amount of data that can be received. For datagram protocols if a packet larger than this is received then excess bytes are irrevocably lost.

The return value is a pair. The `car` is the number of bytes read. The `cdr` is a socket address object (see Section 7.2.11.3 [Network Socket Address], page 533) which is where the data came from, or `#f` if the origin is unknown.

The optional `flags` argument is a or bitwise-OR (`logior`) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Data is read directly from the socket file descriptor, any buffered port data is ignored.

On a GNU/Linux system `recvfrom!` is not multi-threading, all threads stop while a `recvfrom!` call is in progress. An application may need to use `select`, `O_NONBLOCK` or `MSG_DONTWAIT` to avoid this.

sendto sock message sockaddr [flags]  
sendto sock message AF_INET ipv4addr port [flags]  
sendto sock message AF_INET6 ipv6addr port [flowinfo [scopeid [flags]]]  
scm_sendto (sock, message, fam, address, args_and_flags)  
Transmit bytevector `message` as a datagram socket port `sock`. The destination is specified either as a socket address object, or as arguments the same as would be taken by `make-socket-address` to create such an object (see Section 7.2.11.3 [Network Socket Address], page 533).

The destination address may be followed by an optional `flags` argument which is a `logior` (see Section 6.6.2.13 [Bitwise Operations], page 125) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes transmitted – it’s possible for this to be less than the length of `message` if the socket is set to be non-blocking. Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

7.2.11.5 Network Socket Examples

The following give examples of how to use network sockets.

Internet Socket Client Example

The following example demonstrates an Internet socket client. It connects to the HTTP daemon running on the local machine and returns the contents of the root index URL.

```scheme
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (connect s AF_INET (inet-pton AF_INET "127.0.0.1") 80))```
Internet Socket Server Example

The following example shows a simple Internet server which listens on port 2904 for incoming connections and sends a greeting back to the client.

(let ((s (socket PF_INET SOCK_STREAM 0)))
  (setsockopt s SOL_SOCKET SO_REUSEADDR 1)
;; Specific address?
;; (bind s AF_INET (inet-pton AF_INET "127.0.0.1") 2904)
(bind s AF_INET INADDR_ANY 2904)
(listen s 5)

(simple-format #t "Listening for clients in pid: ~S" (getpid))
(newline)

(while #t
  (let* ((client-connection (accept s))
         (client-details (cdr client-connection))
         (client (car client-connection)))
    (simple-format #t "Got new client connection: ~S" client-details)
    (newline)
    (simple-format #t "Client address: ~S"
                   (gethostbyaddr
                    (sockaddr:addr client-details)))
    (newline)
    ;; Send back the greeting to the client port
    (display "Hello client\r\n" client)
    (close client)))))

7.2.12 System Identification

This section lists the various procedures Guile provides for accessing information about the system it runs on.

uname [Scheme Procedure]
scm_uname () [C Function]

Return an object with some information about the computer system the program is running on.

The following procedures accept an object as returned by uname and return a selected component (all of which are strings).
utsname:sysname un
The name of the operating system.

utsname:nodename un
The network name of the computer.

utsname:release un
The current release level of the operating system implementation.

utsname:version un
The current version level within the release of the operating system.

utsname:machine un
A description of the hardware.

gethostname
[Scheme Procedure]
scm_gethostname ()
[C Function]
Return the host name of the current processor.

sethostname name
[Scheme Procedure]
scm_sethostname (name)
[C Function]
Set the host name of the current processor to name. May only be used by the superuser. The return value is not specified.

7.2.13 Locales

setlocale category [locale]
[Scheme Procedure]
scm_setlocale (category, locale)
[C Function]
Get or set the current locale, used for various internationalizations. Locales are strings, such as 'sv_SE'.

If locale is given then the locale for the given category is set and the new value returned. If locale is not given then the current value is returned. category should be one of the following values (see Section “Locale Categories” in The GNU C Library Reference Manual):

LC_ALL
[Variable]
LC_COLLATE
[Variable]
LC_CTYPE
[Variable]
LC_MESSAGES
[Variable]
LC_MONETARY
[Variable]
LC_NUMERIC
[Variable]
LC_TIME
[Variable]
A common usage is ‘(setlocale LC_ALL "")’, which initializes all categories based on standard environment variables (LANG etc). For full details on categories and locale names see Section “Locales and Internationalization” in The GNU C Library Reference Manual.

Note that setlocale affects locale settings for the whole process. See Section 6.25.1 [i18n Introduction], page 459, for a thread-safe alternative.
7.2.14 Encryption

Please note that the procedures in this section are not suited for strong encryption, they are only interfaces to the well-known and common system library functions of the same name. They are just as good (or bad) as the underlying functions, so you should refer to your system documentation before using them (see Section “Encrypting Passwords” in The GNU C Library Reference Manual).

- **crypt** `key salt`  
  [Scheme Procedure]
  Encrypt `key`, with the addition of `salt` (both strings), using the `crypt` C library call.

- **getpass** `prompt`  
  [Scheme Procedure]
  Although **getpass** is not an encryption procedure per se, it appears here because it is often used in combination with **crypt**:

  - **getpass** `prompt`  
    [Scheme Procedure]
    Display `prompt` to the standard error output and read a password from `/dev/tty`. If this file is not accessible, it reads from standard input. The password may be up to 127 characters in length. Additional characters and the terminating newline character are discarded. While reading the password, echoing and the generation of signals by special characters is disabled.

7.3 HTTP, the Web, and All That

It has always been possible to connect computers together and share information between them, but the rise of the World Wide Web over the last couple of decades has made it much easier to do so. The result is a richly connected network of computation, in which Guile forms a part.

By “the web”, we mean the HTTP protocol as handled by servers, clients, proxies, caches, and the various kinds of messages and message components that can be sent and received by that protocol, notably HTML.

On one level, the web is text in motion: the protocols themselves are textual (though the payload may be binary), and it’s possible to create a socket and speak text to the web. But such an approach is obviously primitive. This section details the higher-level data types and operations provided by Guile: URIs, HTTP request and response records, and a conventional web server implementation.

The material in this section is arranged in ascending order, in which later concepts build on previous ones. If you prefer to start with the highest-level perspective, see Section 7.3.10 [Web Examples], page 568, and work your way back.

7.3.1 Types and the Web

It is a truth universally acknowledged, that a program with good use of data types, will be free from many common bugs. Unfortunately, the common practice in web programming seems to ignore this maxim. This subsection makes the case for expressive data types in web programming.

---

2 Yes, the P is for protocol, but this phrase appears repeatedly in RFC 2616.
By “expressive data types”, we mean that the data types say something about how a program solves a problem. For example, if we choose to represent dates using SRFI 19 date records (see Section 7.5.16 [SRFI-19], page 608), this indicates that there is a part of the program that will always have valid dates. Error handling for a number of basic cases, like invalid dates, occurs on the boundary in which we produce a SRFI 19 date record from other types, like strings.

With regards to the web, data types are helpful in the two broad phases of HTTP messages: parsing and generation.

Consider a server, which has to parse a request, and produce a response. Guile will parse the request into an HTTP request object (see Section 7.3.6 [Requests], page 559), with each header parsed into an appropriate Scheme data type. This transition from an incoming stream of characters to typed data is a state change in a program—the strings might parse, or they might not, and something has to happen if they do not. (Guile throws an error in this case.) But after you have the parsed request, “client” code (code built on top of the Guile web framework) will not have to check for syntactic validity. The types already make this information manifest.

This state change on the parsing boundary makes programs more robust, as they themselves are freed from the need to do a number of common error checks, and they can use normal Scheme procedures to handle a request instead of ad-hoc string parsers.

The need for types on the response generation side (in a server) is more subtle, though not less important. Consider the example of a POST handler, which prints out the text that a user submits from a form. Such a handler might include a procedure like this:

```
;; First, a helper procedure
(define (para . contents)
  (string-append "<p>
  " (string-concatenate contents) "</p>"))

;; Now the meat of our simple web application
(define (you-said text)
  (para "You said: " text))

(display (you-said "Hi!"))
⊢ <p>You said: Hi!</p>
```

This is a perfectly valid implementation, provided that the incoming text does not contain the special HTML characters ‘<’, ‘>’, or ‘&’. But this provision of a restricted character set is not reflected anywhere in the program itself: we must assume that the programmer understands this, and performs the check elsewhere.

Unfortunately, the short history of the practice of programming does not bear out this assumption. A cross-site scripting (XSS) vulnerability is just such a common error in which unfiltered user input is allowed into the output. A user could submit a crafted comment to your web site which results in visitors running malicious Javascript, within the security context of your domain:

```
(display (you-said "<script src="http://bad.com/nasty.js" />
))
⊢ <p>You said: <script src="http://bad.com/nasty.js" /></p>
```
The fundamental problem here is that both user data and the program template are represented using strings. This identity means that types can’t help the programmer to make a distinction between these two, so they get confused.

There are a number of possible solutions, but perhaps the best is to treat HTML not as strings, but as native s-expressions: as SXML. The basic idea is that HTML is either text, represented by a string, or an element, represented as a tagged list. So ‘foo’ becomes ‘"foo"’, and ‘<b>foo</b>’ becomes ‘(b "foo”)’. Attributes, if present, go in a tagged list headed by ‘@’, like ‘(img (@ (src "http://example.com/foo.png")))’. See Section 7.20 [SXML], page 733, for more information.

The good thing about SXML is that HTML elements cannot be confused with text. Let’s make a new definition of para:

```scheme
(define (para . contents)
  '(p ,@contents))
```

```scheme
(use-modules (sxml simple))
(sxml->xml (you-said "Hi!"))
⊢ <p>You said: Hi!</p>
```

```scheme
(sxml->xml (you-said "&lt;i&gt;Rats, foiled again!&lt;/i&gt;"))
⊢ <p>You said: &lt;i&gt;Rats, foiled again!&lt;/i&gt;&lt;/p&gt;
```

So we see in the second example that HTML elements cannot be unwittingly introduced into the output. However it is now perfectly acceptable to pass SXML to `you-said`; in fact, that is the big advantage of SXML over everything-as-a-string.

```scheme
(sxml->xml (you-said (you-said "&lt;Hi!&gt;")))
⊢ <p>You said: <p>You said: &lt;Hi!&gt;&lt;/p&gt;
```

The SXML types allow procedures to compose. The types make manifest which parts are HTML elements, and which are text. So you needn’t worry about escaping user input; the type transition back to a string handles that for you. XSS vulnerabilities are a thing of the past.

Well. That’s all very nice and opinionated and such, but how do I use the thing? Read on!

### 7.3.2 Universal Resource Identifiers

Guile provides a standard data type for Universal Resource Identifiers (URIs), as defined in RFC 3986.

The generic URI syntax is as follows:

```
URI-reference := [scheme ":" ["//" [userinfo "@"] host [":" port]] path \ [ "?" query ] [ "#" fragment ]
```

For example, in the URI, `http://www.gnu.org/help/`, the scheme is `http`, the host is `www.gnu.org`, the path is `/help/`, and there is no userinfo, port, query, or fragment.

Userinfo is something of an abstraction, as some legacy URI schemes allowed userinfo of the form `username:passwd`. But since passwords do not belong in URIs, the RFC does not want to condone this practice, so it calls anything before the @ sign `userinfo`.

```scheme
(use-modules (web uri))
```
The following procedures can be found in the (web uri) module. Load it into your Guile, using a form like the above, to have access to them.

The most common way to build a URI from Scheme is with the build-uri function.

```scheme
(build-uri scheme [#:userinfo=#f] [#:host=#f] [#:port=#f] [#:path=""] [#:query=#f] [#:fragment=#f] [#:validate?=#t])
```

Construct a URI. scheme should be a symbol, port either a positive, exact integer or #f, and the rest of the fields are either strings or #f. If validate? is true, also run some consistency checks to make sure that the constructed URI is valid.

```scheme
(uri? obj)
```

Return #t if obj is a URI.

Guile, URIs are represented as URI records, with a number of associated accessors.

```scheme
(uri-scheme uri)                  ; [Scheme Procedure]
(uri-userinfo uri)               ; [Scheme Procedure]
(uri-host uri)                   ; [Scheme Procedure]
(uri-port uri)                   ; [Scheme Procedure]
(uri-path uri)                   ; [Scheme Procedure]
(uri-query uri)                  ; [Scheme Procedure]
(uri-fragment uri)               ; [Scheme Procedure]
```

Field accessors for the URI record type. The URI scheme will be a symbol, or #f if the object is a relative-ref (see below). The port will be either a positive, exact integer or #f, and the rest of the fields will be either strings or #f if not present.

```scheme
(string->uri string)             ; [Scheme Procedure]
```

Parse string into a URI object. Return #f if the string could not be parsed.

```scheme
(uri->string uri [#:include-fragment?=#t]) ; [Scheme Procedure]
```

Serialize uri to a string. If the URI has a port that is the default port for its scheme, the port is not included in the serialization. If include-fragment? is given as false, the resulting string will omit the fragment (if any).

```scheme
(declare-default-port! scheme port) ; [Scheme Procedure]
```

Declare a default port for the given URI scheme.

```scheme
(uri-decode str [#:encoding="utf-8"] [#:decode-plus-to-space? #t]) ; [Scheme Procedure]
```

Percent-decode the given str, according to encoding, which should be the name of a character encoding.

Note that this function should not generally be applied to a full URI string. For paths, use split-and-decode-uri-path instead. For query strings, split the query on & and = boundaries, and decode the components separately.

Note also that percent-encoded strings encode bytes, not characters. There is no guarantee that a given byte sequence is a valid string encoding. Therefore this routine may signal an error if the decoded bytes are not valid for the given encoding. Pass #f for encoding if you want decoded bytes as a bytevector directly. See Section 6.14.1 [Ports], page 328, for more information on character encodings.
If `decode-plus-to-space?` is true, which is the default, also replace instances of the plus character ‘+’ with a space character. This is needed when parsing `application/x-www-form-urlencoded` data.

Returns a string of the decoded characters, or a bytevector if `encoding` was `#f`.

`uri-encode str [#:encoding="utf-8"] [#:unescape-chars]` [Scheme Procedure]
Percent-encode any character not in the character set, `unescape-chars`.

The default character set includes alphanumerics from ASCII, as well as the special characters ‘-’, ‘.’, ‘_’, and ‘~’. Any other character will be percent-encoded, by writing out the character to a bytevector within the given `encoding`, then encoding each byte as `%HH`, where `HH` is the hexadecimal representation of the byte.

`split-and-decode-uri-path path` [Scheme Procedure]
Split `path` into its components, and decode each component, removing empty components.

For example, `"/foo/bar%20baz/"` decodes to the two-element list, `("foo" "bar baz")`.

`encode-and-join-uri-path parts` [Scheme Procedure]
URI-encode each element of `parts`, which should be a list of strings, and join the parts together with `/` as a delimiter.

For example, the list `("scrambled eggs" "biscuits&gravy")` encodes as `"scrambled%20eggs/biscuits%26gravy"`.

### Subtypes of URI

As we noted above, not all URI objects have a scheme. You might have noted in the “generic URI syntax” example that the left-hand side of that grammar definition was URI-reference, not URI. A `URI-reference` is a generalization of a URI where the scheme is optional. If no scheme is specified, it is taken to be relative to some other related URI. A common use of URI references is when you want to be vague regarding the choice of HTTP or HTTPS – serving a web page referring to `/foo.css` will use HTTPS if loaded over HTTPS, or HTTP otherwise.

`build-uri-reference [#:scheme=#f] [#:userinfo=#f]` [Scheme Procedure]

Like `build-uri`, but with an optional scheme.

`uri-reference? obj` [Scheme Procedure]
Return `#t` if `obj` is a URI-reference. This is the most general URI predicate, as it includes not only full URIs that have schemes (those that match `uri?`) but also URIs without schemes.

It’s also possible to build a `relative-ref`: a URI-reference that explicitly lacks a scheme.

`build-relative-ref [#:userinfo=#f] [#:host=#f]` [Scheme Procedure]

Like `build-uri`, but with no scheme.
**relative-ref? obj**  
[Scheme Procedure]
Return `#t` if `obj` is a “relative-ref”: a URI-reference that has no scheme. Every URI-reference will either match `uri?` or `relative-ref?` (but not both).

In case it’s not clear from the above, the most general of these URI types is the URI-reference, with `build-uri-reference` as the most general constructor. `build-uri` and `build-relative-ref` enforce enforce specific restrictions on the URI-reference. The most generic URI parser is then `string->uri-reference`, and there is also a parser for when you know that you want a relative-ref.

**string->uri-reference string**  
[Scheme Procedure]
Parse `string` into a URI object, while not requiring a scheme. Return `#f` if the string could not be parsed.

**string->relative-ref string**  
[Scheme Procedure]
Parse `string` into a URI object, while asserting that no scheme is present. Return `#f` if the string could not be parsed.

For compatibility reasons, note that `uri?` will return `#t` for all URI objects, even relative-refs. In contrast, `build-uri` and `string->uri` require that the resulting URI not be a relative-ref. As a predicate to distinguish relative-refs from proper URIs (in the language of RFC 3986), use something like `(and (uri-reference? x) (not (relative-ref? x)))`.

### 7.3.3 The Hyper-Text Transfer Protocol

The initial motivation for including web functionality in Guile, rather than rely on an external package, was to establish a standard base on which people can share code. To that end, we continue the focus on data types by providing a number of low-level parsers and unparsers for elements of the HTTP protocol.

If you are want to skip the low-level details for now and move on to web pages, see Section 7.3.8 [Web Client], page 564, and see Section 7.3.9 [Web Server], page 565. Otherwise, load the HTTP module, and read on.

```
(use-modules (web http))
```

The focus of the `(web http)` module is to parse and unparses standard HTTP headers, representing them to Guile as native data structures. For example, a `Date:` header will be represented as a SRFI-19 date record (see Section 7.5.16 [SRFI-19], page 608), rather than as a string.

Guile tries to follow RFCs fairly strictly—the road to perdition being paved with compatibility hacks—though some allowances are made for not-too-divergent texts.

Header names are represented as lower-case symbols.

**string->header name**  
[Scheme Procedure]
Parse `name` to a symbolic header name.

**header->string sym**  
[Scheme Procedure]
Return the string form for the header named `sym`. 

For example:

(string->header "Content-Length")
⇒ content-length
(header->string 'content-length)
⇒ "Content-Length"

(string->header "FOO")
⇒ foo
(header->string 'foo)
⇒ "Foo"

Guile keeps a registry of known headers, their string names, and some parsing and serialization procedures. If a header is unknown, its string name is simply its symbol name in title-case.

known-header? sym

Return #t if sym is a known header, with associated parsers and serialization procedures, or #f otherwise.

header-parser sym

Return the value parser for headers named sym. The result is a procedure that takes one argument, a string, and returns the parsed value. If the header isn’t known to Guile, a default parser is returned that passes through the string unchanged.

header-validator sym

Return a predicate which returns #t if the given value is valid for headers named sym. The default validator for unknown headers is string?.

header-writer sym

Return a procedure that writes values for headers named sym to a port. The resulting procedure takes two arguments: a value and a port. The default writer is display.

For more on the set of headers that Guile knows about out of the box, see Section 7.3.4 [HTTP Headers], page 551. To add your own, use the declare-header! procedure:

declare-header! name parser validator writer
  [#:multiple?=#f]

Declare a parser, validator, and writer for a given header.

For example, let’s say you are running a web server behind some sort of proxy, and your proxy adds an X-Client-Address header, indicating the IPv4 address of the original client. You would like for the HTTP request record to parse out this header to a Scheme value, instead of leaving it as a string. You could register this header with Guile’s HTTP stack like this:

(declare-header! "X-Client-Address"
  (lambda (str)
    (inet-pton AF_INET str))
  (lambda (ip)
    (and (integer? ip) (exact? ip) (<= 0 ip #xffffffff)))
  (lambda (ip port)
    (display (inet-ntop AF_INET ip) port)))
**Declare-opaque-header! name**

A specialised version of `declare-header!` for the case in which you want a header’s value to be returned/written “as-is”.

**Valid-header? sym val**

Return a true value if `val` is a valid Scheme value for the header with name `sym`, or `#f` otherwise.

Now that we have a generic interface for reading and writing headers, we do just that.

**Read-header port**

Read one HTTP header from `port`. Return two values: the header name and the parsed Scheme value. May raise an exception if the header was known but the value was invalid.

Returns the end-of-file object for both values if the end of the message body was reached (i.e., a blank line).

**Parse-header name val**

Parse `val`, a string, with the parser for the header named `name`. Returns the parsed value.

**Write-header name val port**

Write the given header name and value to `port`, using the writer from `header-writer`.

**Read-headers port**

Read the headers of an HTTP message from `port`, returning them as an ordered alist.

**Write-headers headers port**

Write the given header alist to `port`. Doesn’t write the final `\r\n`, as the user might want to add another header.

The `(web http)` module also has some utility procedures to read and write request and response lines.

**Parse-http-method str [start] [end]**

Parse an HTTP method from `str`. The result is an upper-case symbol, like `GET`.

**Parse-http-version str [start] [end]**

Parse an HTTP version from `str`, returning it as a major–minor pair. For example, `HTTP/1.1` parses as the pair of integers, `(1 . 1)`.

**Parse-request-uri str [start] [end]**

Parse a URI from an HTTP request line. Note that URIs in requests do not have to have a scheme or host name. The result is a URI object.

**Read-request-line port**

Read the first line of an HTTP request from `port`, returning three values: the method, the URI, and the version.

**Write-request-line method uri version port**

Write the first line of an HTTP request to `port`. 
read-response-line port

Read the first line of an HTTP response from port, returning three values: the HTTP version, the response code, and the “reason phrase”.

write-response-line version code reason-phrase port

Write the first line of an HTTP response to port.

7.3.4 HTTP Headers

In addition to defining the infrastructure to parse headers, the (web http) module defines specific parsers and unparsers for all headers defined in the HTTP/1.1 standard.

For example, if you receive a header named ‘Accept-Language’ with a value ‘en, es;q=0.8’, Guile parses it as a quality list (defined below):

(parse-header 'accept-language "en, es;q=0.8")
⇒ ((1000 . "en") (800 . "es"))

The format of the value for ‘Accept-Language’ headers is defined below, along with all other headers defined in the HTTP standard. (If the header were unknown, the value would have been returned as a string.)

For brevity, the header definitions below are given in the form, Type name, indicating that values for the header name will be of the given Type. Since Guile internally treats header names in lower case, in this document we give types title-cased names. A short description of the each header’s purpose and an example follow.

For full details on the meanings of all of these headers, see the HTTP 1.1 standard, RFC 2616.

7.3.4.1 HTTP Header Types

Here we define the types that are used below, when defining headers.

Date
A SRFI-19 date.

KVList
A list whose elements are keys or key-value pairs. Keys are parsed to symbols. Values are strings by default. Non-string values are the exception, and are mentioned explicitly below, as appropriate.

SList
A list of strings.

Quality
An exact integer between 0 and 1000. Qualities are used to express preference, given multiple options. An option with a quality of 870, for example, is preferred over an option with quality 500.

(qualities are written out over the wire as numbers between 0.0 and 1.0, but since the standard only allows three digits after the decimal, it’s equivalent to integers between 0 and 1000, so that’s what Guile uses.)
QList

A quality list: a list of pairs, the car of which is a quality, and the cdr a string. Used to express a list of options, along with their qualities.

ETag

An entity tag, represented as a pair. The car of the pair is an opaque string, and the cdr is #t if the entity tag is a “strong” entity tag, and #f otherwise.

7.3.4.2 General Headers

General HTTP headers may be present in any HTTP message.

KVList cache-control

A key-value list of cache-control directives. See RFC 2616, for more details.

If present, parameters to max-age, max-stale, min-fresh, and s-maxage are all parsed as non-negative integers.

If present, parameters to private and no-cache are parsed as lists of header names, as symbols.

(parse-header 'cache-control "no-cache,no-store"
⇒ (no-cache no-store)
(parse-header 'cache-control "no-cache="Authorization,Date",no-store"
⇒ ((no-cache . (authorization date)) no-store)
(parse-header 'cache-control "no-cache="Authorization,Date",max-age=10"
⇒ ((no-cache . (authorization date)) (max-age . 10))

List connection

A list of header names that apply only to this HTTP connection, as symbols. Additionally, the symbol 'close' may be present, to indicate that the server should close the connection after responding to the request.

(parse-header 'connection "close"
⇒ (close)

Date date

The date that a given HTTP message was originated.

(parse-header 'date "Tue, 15 Nov 1994 08:12:31 GMT"
⇒ #<date ...>}

KVList pragma

A key-value list of implementation-specific directives.

(parse-header 'pragma "no-cache, broccoli=tasty"
⇒ (no-cache (broccoli . "tasty"))

List trailer

A list of header names which will appear after the message body, instead of with the message headers.

(parse-header 'trailer "ETag"
⇒ (etag)
List **transfer-encoding**

A list of transfer codings, expressed as key-value lists. The only transfer coding defined by the specification is `chunked`.

```
(parse-header 'transfer-encoding "chunked")
⇒ ((chunked))
```

List **upgrade**

A list of strings, indicating additional protocols that a server could use in response to a request.

```
(parse-header 'upgrade "WebSocket")
⇒ ("WebSocket")
```

FIXME: parse out more fully?

List **via**

A list of strings, indicating the protocol versions and hosts of intermediate servers and proxies. There may be multiple `via` headers in one message.

```
(parse-header 'via "1.0 venus, 1.1 mars")
⇒ ("1.0 venus" "1.1 mars")
```

List **warning**

A list of warnings given by a server or intermediate proxy. Each warning is itself a list of four elements: a code, as an exact integer between 0 and 1000, a host as a string, the warning text as a string, and either `#f` or a SRFI-19 date.

There may be multiple `warning` headers in one message.

```
(parse-header 'warning "123 foo "core breach imminent""")
⇒ ((123 "foo" "core-breach imminent" #f))
```

### 7.3.4.3 Entity Headers

Entity headers may be present in any HTTP message, and refer to the resource referenced in the HTTP request or response.

List **allow**

A list of allowed methods on a given resource, as symbols.

```
(parse-header 'allow "GET, HEAD")
⇒ (GET HEAD)
```

List **content-encoding**

A list of content codings, as symbols.

```
(parse-header 'content-encoding "gzip")
⇒ (gzip)
```

List **content-language**

The languages that a resource is in, as strings.

```
(parse-header 'content-language "en")
⇒ ("en")
```
UI Int content-length [HTTP Header]
The number of bytes in a resource, as an exact, non-negative integer.

(parse-header 'content-length "300")
⇒ 300

URI content-location [HTTP Header]
The canonical URI for a resource, in the case that it is also accessible from a different URI.

(parse-header 'content-location "http://example.com/foo")
⇒ #<<uri> ...>

String content-md5 [HTTP Header]
The MD5 digest of a resource.

(parse-header 'content-md5 "ffaea1a79810785575e29e2bd45e2fa5")
⇒ "ffaea1a79810785575e29e2bd45e2fa5"

List content-range [HTTP Header]
A range specification, as a list of three elements: the symbol bytes, either the symbol * or a pair of integers, indicating the byte rage, and either * or an integer, for the instance length. Used to indicate that a response only includes part of a resource.

(parse-header 'content-range "bytes 10-20/*")
⇒ (bytes (10 . 20) *)

List content-type [HTTP Header]
The MIME type of a resource, as a symbol, along with any parameters.

(parse-header 'content-type "text/plain")
⇒ (text/plain)
(parse-header 'content-type "text/plain; charset=utf-8")
⇒ (text/plain (charset "utf-8"))

Note that the charset parameter is something of a misnomer, and the HTTP specification admits this. It specifies the encoding of the characters, not the character set.

Date expires [HTTP Header]
The date/time after which the resource given in a response is considered stale.

(parse-header 'expires "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>

Date last-modified [HTTP Header]
The date/time on which the resource given in a response was last modified.

(parse-header 'expires "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>

7.3.4.4 Request Headers
Request headers may only appear in an HTTP request, not in a response.
List accept

A list of preferred media types for a response. Each element of the list is itself a list, in the same format as content-type.

(parse-header 'accept "text/html,text/plain;charset=utf-8")
⇒ ((text/html) (text/plain (charset . "utf-8")))

Preference is expressed with quality values:

(parse-header 'accept "text/html;q=0.8,text/plain;q=0.6")
⇒ ((text/html (q . 800)) (text/plain (q . 600)))

QList accept-charset

A quality list of acceptable charsets. Note again that what HTTP calls a “charset” is what Guile calls a “character encoding”.

(parse-header 'accept-charset "iso-8859-5, unicode-1-1;q=0.8")
⇒ ((1000 . "iso-8859-5") (800 . "unicode-1-1"))

QList accept-encoding

A quality list of acceptable content codings.

(parse-header 'accept-encoding "gzip,identity=0.8")
⇒ ((1000 . "gzip") (800 . "identity"))

QList accept-language

A quality list of acceptable languages.

(parse-header 'accept-language "cn,en=0.75")
⇒ ((1000 . "cn") (750 . "en"))

Pair authorization

Authorization credentials. The car of the pair indicates the authentication scheme, like basic. For basic authentication, the cdr of the pair will be the base64-encoded ‘user:pass’ string. For other authentication schemes, like digest, the cdr will be a key-value list of credentials.

(parse-header 'authorization "Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
⇒ (basic . "QWxhZGRpbjpvcGVuIHNlc2FtZQ==")

List expect

A list of expectations that a client has of a server. The expectations are key-value lists.

(parse-header 'expect "100-continue")
⇒ ((100-continue))

String from

The email address of a user making an HTTP request.

(parse-header 'from "bob@example.com")
⇒ "bob@example.com"

Pair host

The host for the resource being requested, as a hostname-port pair. If no port is given, the port is #f.

(parse-header 'host "gnu.org:80")
⇒ ("gnu.org" . 80)
⇒ (parse-header 'host "gnu.org")
⇒ ("gnu.org" . #f)

*|List if-match

A set of etags, indicating that the request should proceed if and only if the etag of
the resource is in that set. Either the symbol *, indicating any etag, or a list of entity
tags.

⇒ *
⇒ (("asdfadf" . #t))
⇒ (("asdfadf" . #f))

Date if-modified-since

Indicates that a response should proceed if and only if the resource has been modified
since the given date.

⇒ #<date ...

*|List if-none-match

A set of etags, indicating that the request should proceed if and only if the etag of
the resource is not in the set. Either the symbol *, indicating any etag, or a list of
entity tags.

⇒ *

ETag|Date if-range

Indicates that the range request should proceed if and only if the resource matches a
modification date or an etag. Either an entity tag, or a SRFI-19 date.

⇒ ("original-etag" . #t)

Date if-unmodified-since

Indicates that a response should proceed if and only if the resource has not been
modified since the given date.

⇒ #<date ...

UInt max-forwards

The maximum number of proxy or gateway hops that a request should be subject to.

⇒ 10

Pair proxy-authorization

Authorization credentials for a proxy connection. See the documentation for
authorization above for more information on the format.

⇒ (parse-header 'proxy-authorization "Digest foo=bar,baz=qux"
⇒ (digest (foo . "bar") (baz . "qux"))

Pair range
A range request, indicating that the client wants only part of a resource. The car
of the pair is the symbol bytes, and the cdr is a list of pairs. Each element of the
cdr indicates a range; the car is the first byte position and the cdr is the last byte
position, as integers, or #f if not given.

⇒ (parse-header 'range "bytes=10-30,50-")
⇒ (bytes (10 . 30) (50 . #f))

URI referer
The URI of the resource that referred the user to this resource. The name of the
header is a misspelling, but we are stuck with it.

⇒ (parse-header 'referer "http://www.gnu.org/")
⇒ #<uri ...>

List te
A list of transfer codings, expressed as key-value lists. A common transfer coding is
trailers.

⇒ ((trailers))

String user-agent
A string indicating the user agent making the request. The specification defines a
structured format for this header, but it is widely disregarded, so Guile does not
attempt to parse strictly.

⇒ "Mozilla/5.0"

7.3.4.5 Response Headers

List accept-ranges
A list of range units that the server supports, as symbols.

⇒ (parse-header 'accept-ranges "bytes")
⇒ (bytes)

UInt age
The age of a cached response, in seconds.

⇒ (parse-header 'age "3600")
⇒ 3600

ETag etag
The entity-tag of the resource.

⇒ (parse-header 'etag "\"foo\""")
⇒ ("foo" . #t)
URI-reference location  
A URI reference on which a request may be completed. Used in combination with a redirecting status code to perform client-side redirection.

```lisp
(parse-header 'location "http://example.com/other")
⇒ #<uri ...>
```

List proxy-authenticate  
A list of challenges to a proxy, indicating the need for authentication.

```lisp
(parse-header 'proxy-authenticate "Basic realm="foo""
⇒ ((basic (realm . "foo")))
```

UInt|Date retry-after  
Used in combination with a server-busy status code, like 503, to indicate that a client should retry later. Either a number of seconds, or a date.

```lisp
(parse-header 'retry-after "60")
⇒ 60
```

String server  
A string identifying the server.

```lisp
(parse-header 'server "My first web server")
⇒ "My first web server"
```

*|List vary  
A set of request headers that were used in computing this response. Used to indicate that server-side content negotiation was performed, for example in response to the accept-language header. Can also be the symbol *, indicating that all headers were considered.

```lisp
(parse-header 'vary "Accept-Language, Accept"
⇒ (accept-language accept)
```

List www-authenticate  
A list of challenges to a user, indicating the need for authentication.

```lisp
(parse-header 'www-authenticate "Basic realm="foo""
⇒ ((basic (realm . "foo")))
```

### 7.3.5 Transfer Codings

HTTP 1.1 allows for various transfer codings to be applied to message bodies. These include various types of compression, and HTTP chunked encoding. Currently, only chunked encoding is supported by guile.

Chunked coding is an optional coding that may be applied to message bodies, to allow messages whose length is not known beforehand to be returned. Such messages can be split into chunks, terminated by a final zero length chunk.

In order to make dealing with encodings more simple, guile provides procedures to create ports that “wrap” existing ports, applying transformations transparently under the hood.

These procedures are in the (web http) module.

```lisp
(use-modules (web http))
```
**make-chunked-input-port** port [#:keep-alive?=#:f]  
Returns a new port, that transparently reads and decodes chunk-encoded data from *port*. If no more chunk-encoded data is available, it returns the end-of-file object. When the port is closed, *port* will also be closed, unless *keep-alive*? is true.

```
(use-modules (ice-9 rdelim))
(define s "5\r\nFirst\r\nA\n line
Sec\r\nond line\r\n0\r")
(define p (make-chunked-input-port (open-input-string s)))
(read-line s)
⇒ "First line"
(read-line s)
⇒ "Second line"
```

**make-chunked-output-port** port [#:keep-alive?=#:f]  
Returns a new port, which transparently encodes data as chunk-encoded before writing it to *port*. Whenever a write occurs on this port, it buffers it, until the port is flushed, at which point it writes a chunk containing all the data written so far. When the port is closed, the data remaining is written to *port*, as is the terminating zero chunk. It also causes *port* to be closed, unless *keep-alive*? is true.

Note. Forcing a chunked output port when there is no data is buffered does not write a zero chunk, as this would cause the data to be interpreted incorrectly by the client.

```
(call-with-output-string
  (lambda (out)
    (define out* (make-chunked-output-port out #:keep-alive? #t))
    (display "first chunk" out*)
    (force-output out*)
    (force-output out*); note this does not write a zero chunk
    (display "second chunk" out*)
    (close-port out*))
⇒ "b\r\nfirst chunk\r\nc\r
second chunk\r\n0\r"
```

### 7.3.6 HTTP Requests

The request module contains a data type for HTTP requests.

#### 7.3.6.1 An Important Note on Character Sets

HTTP requests consist of two parts: the request proper, consisting of a request line and a set of headers, and (optionally) a body. The body might have a binary content-type, and even in the textual case its length is specified in bytes, not characters.

Therefore, HTTP is a fundamentally binary protocol. However the request line and headers are specified to be in a subset of ASCII, so they can be treated as text, provided that the port’s encoding is set to an ASCII-compatible one-byte-per-character encoding. ISO-8859-1 (latin-1) is just such an encoding, and happens to be very efficient for Guile.

So what Guile does when reading requests from the wire, or writing them out, is to set the port’s encoding to latin-1, and treating the request headers as text.
The request body is another issue. For binary data, the data is probably in a bytevector, so we use the R6RS binary output procedures to write out the binary payload. Textual data usually has to be written out to some character encoding, usually UTF-8, and then the resulting bytevector is written out to the port.

In summary, Guile reads and writes HTTP over latin-1 sockets, without any loss of generality.

### 7.3.6.2 Request API

- `request? obj` [Scheme Procedure]
- `request-method request` [Scheme Procedure]
- `request-uri request` [Scheme Procedure]
- `request-version request` [Scheme Procedure]
- `request-headers request` [Scheme Procedure]
- `request-meta request` [Scheme Procedure]
- `request-port request` [Scheme Procedure]

A predicate and field accessors for the request type. The fields are as follows:

- **method** The HTTP method, for example, `GET`.
- **uri** The URI as a URI record.
- **version** The HTTP version pair, like `(1 . 1)`.
- **headers** The request headers, as an alist of parsed values.
- **meta** An arbitrary alist of other data, for example information returned in the `sockaddr` from `accept` (see Section 7.2.11.4 [Network Sockets and Communication], page 535).
- **port** The port on which to read or write a request body, if any.

- `read-request port [meta='']` [Scheme Procedure]

  Read an HTTP request from `port`, optionally attaching the given metadata, `meta`. As a side effect, sets the encoding on `port` to ISO-8859-1 (latin-1), so that reading one character reads one byte. See the discussion of character sets above, for more information.

  Note that the body is not part of the request. Once you have read a request, you may read the body separately, and likewise for writing requests.

- `build-request uri [#:method='GET'] [#:version=('1 . 1')]` [Scheme Procedure]

  Construct an HTTP request object. If `validate-headers?` is true, the headers are each run through their respective validators.

- `write-request r port` [Scheme Procedure]

  Write the given HTTP request to `port`.

  Return a new request, whose `request-port` will continue writing on `port`, perhaps using some transfer encoding.
Chapter 7: Guile Modules

read-request-body r

Reads the request body from r, as a bytevector. Return #f if there was no request body.

write-request-body r bv

Write bv, a bytevector, to the port corresponding to the HTTP request r.

The various headers that are typically associated with HTTP requests may be accessed with these dedicated accessors. See Section 7.3.4 [HTTP Headers], page 551, for more information on the format of parsed headers.

request-accept request [default='()]
request-accept-charset request [default='()]
request-accept-encoding request [default='()]
request-accept-language request [default='()]
request-allow request [default='()]
request-authorization request [default=#f]
request-cache-control request [default='()]
request-connection request [default='()]
request-content-encoding request [default='()]
request-content-language request [default='()]
request-content-length request [default=#f]
request-content-location request [default=#f]
request-content-md5 request [default=#f]
request-content-range request [default=#f]
request-content-type request [default=#f]
request-date request [default=#f]
request-expect request [default='()]
request-expires request [default=#f]
request-from request [default=#f]
request-host request [default=#f]
request-if-match request [default=#f]
request-if-modified-since request [default=#f]
request-if-none-match request [default=#f]
request-if-range request [default=#f]
request-if-unmodified-since request [default=#f]
request-last-modified request [default=#f]
request-max-forwards request [default=#f]
request-pragma request [default='()]
request-proxy-authorization request [default=#f]
request-range request [default=#f]
request-referer request [default=#f]
request-te request [default=#f]
request-trailer request [default='()]
request-transfer-encoding request [default='()]
request-upgrade request [default='()]
request-user-agent request [default=#f]
request-via request [default='()]

[Scheme Procedure]
request-warning request [default='()]                  [Scheme Procedure]
   Return the given request header, or default if none was present.

request-absolute-uri r [default-host=#f]          [Scheme Procedure]
   [default-port=#f] [default-scheme=#f]
   A helper routine to determine the absolute URI of a request, using the host header and the default scheme, host and port. If there is no default scheme and the URI is not itself absolute, an error is signalled.

7.3.7 HTTP Responses

   (use-modules (web response))

   As with requests (see Section 7.3.6 [Requests], page 559), Guile offers a data type for HTTP responses. Again, the body is represented separately from the request.

response? obj                        [Scheme Procedure]
response-version response            [Scheme Procedure]
response-code response               [Scheme Procedure]
response-reason-phrase response      [Scheme Procedure]
response-headers response           [Scheme Procedure]
response-port response              [Scheme Procedure]

   A predicate and field accessors for the response type. The fields are as follows:

   version The HTTP version pair, like (1 . 1).
   code The HTTP response code, like 200.
   reason-phrase The reason phrase, or the standard reason phrase for the response’s code.
   headers The response headers, as an alist of parsed values.
   port The port on which to read or write a response body, if any.

read-response port                      [Scheme Procedure]
   Read an HTTP response from port.
   As a side effect, sets the encoding on port to ISO-8859-1 (latin-1), so that reading one character reads one byte. See the discussion of character sets in Section 7.3.7 [Responses], page 562, for more information.

build-response [#:version='(1 . 1)] [#:code=200]
   [#:reason-phrase=#f] [#:headers='()] [#:port=#f]
   [#:validate-headers?=#t]
   Construct an HTTP response object. If validate-headers? is true, the headers are each run through their respective validators.

adapt-response-version response version    [Scheme Procedure]
   Adapt the given response to a different HTTP version. Return a new HTTP response.
   The idea is that many applications might just build a response for the default HTTP version, and this method could handle a number of programmatic transformations to respond to older HTTP versions (0.9 and 1.0). But currently this function is a bit heavy-handed, just updating the version field.
write-response r port  
Write the given HTTP response to port.

Return a new response, whose response-port will continue writing on port, perhaps using some transfer encoding.

response-must-not-include-body? r  
Some responses, like those with status code 304, are specified as never having bodies. This predicate returns #t for those responses.

Note also, though, that responses to HEAD requests must also not have a body.

response-body-port r [#:decode?=#f] [#:keep-alive?=#f]  
Return an input port from which the body of r can be read. The encoding of the returned port is set according to r’s content-type header, when it’s textual, except if decode? is #f. Return #f when no body is available.

When keep-alive? is #f, closing the returned port also closes r’s response port.

read-response-body r  
Read the response body from r, as a bytevector. Returns #f if there was no response body.

write-response-body r bv  
Write bv, a bytevector, to the port corresponding to the HTTP response r.

As with requests, the various headers that are typically associated with HTTP responses may be accessed with these dedicated accessors. See Section 7.3.4 [HTTP Headers], page 551, for more information on the format of parsed headers.
response-trailer response [default=""]
response-transfer-encoding response [default=""]
response-upgrade response [default=""]
response-vary response [default=""]
response-via response [default=""]
response-warning response [default=""]
response-www-authenticate response [default=\#f]

Return the given response header, or default if none was present.

text-content-type? type

Return #t if type, a symbol as returned by response-content-type, represents a textual type such as text/plain.

7.3.8 Web Client

(web client) provides a simple, synchronous HTTP client, built on the lower-level HTTP, request, and response modules.

(use-modules (web client))

open-socket-for-uri uri

Return an open input/output port for a connection to URI. Guile dynamically loads GnuTLS for HTTPS support. See Section “Guile Preparations” in GnuTLS-Guile, for more information.

http-request uri arg...

Connect to the server corresponding to uri and make a request over HTTP, using method (GET, HEAD, POST, etc.).

The following keyword arguments allow you to modify the requests in various ways, for example attaching a body to the request, or setting specific headers. The following table lists the keyword arguments and their default values.

#:method 'GET
#:body #f
#:port (open-socket-for-uri uri)]
#:version '(1 . 1)
#:keep-alive? #f
#:headers '()
#:decode-body? #t
#:streaming? #f

If you already have a port open, pass it as port. Otherwise, a connection will be opened to the server corresponding to uri. Any extra headers in the alist headers will be added to the request.

If body is not #f, a message body will also be sent with the HTTP request. If body is a string, it is encoded according to the content-type in headers, defaulting to UTF-8. Otherwise body should be a bytevector, or #f for no body. Although a message body may be sent with any request, usually only POST and PUT requests have bodies.

If decode-body? is true, as is the default, the body of the response will be decoded to string, if it is a textual content-type. Otherwise it will be returned as a bytevector.
However, if `streaming?` is true, instead of eagerly reading the response body from the server, this function only reads off the headers. The response body will be returned as a port on which the data may be read.

Unless `keep-alive?` is true, the port will be closed after the full response body has been read.

Returns two values: the response read from the server, and the response body as a string, bytevector, #f value, or as a port (if `streaming?` is true).

```
http-get uri arg... [Scheme Procedure]
http-head uri arg... [Scheme Procedure]
http-post uri arg... [Scheme Procedure]
http-put uri arg... [Scheme Procedure]
http-delete uri arg... [Scheme Procedure]
http-trace uri arg... [Scheme Procedure]
http-options uri arg... [Scheme Procedure]
```

Connect to the server corresponding to `uri` and make a request over HTTP, using the appropriate method (`GET`, `HEAD`, `POST`, etc.).

These procedures are variants of `http-request` specialized with a specific `method` argument, and have the same prototype: a URI followed by an optional sequence of keyword arguments. See [http-request], page 564, for full documentation on the various keyword arguments.

`http-get` is useful for making one-off requests to web sites. If you are writing a web spider or some other client that needs to handle a number of requests in parallel, it’s better to build an event-driven URL fetcher, similar in structure to the web server (see Section 7.3.9 [Web Server], page 565).

Another option, good but not as performant, would be to use threads, possibly via `par-map` or futures.

```
current-http-proxy [Scheme Parameter]
```

Either `#f` or a non-empty string containing the URL of the HTTP proxy server to be used by the procedures in the `(web client)` module, including `open-socket-for-uri`. Its initial value is based on the `http_proxy` environment variable.

```
(current-http-proxy) ⇒ "http://localhost:8123/
(parameterize ((current-http-proxy #f))
  (http-get "http://example.com/") ) ; temporarily bypass proxy
(current-http-proxy) ⇒ "http://localhost:8123/
```

### 7.3.9 Web Server

`(web server)` is a generic web server interface, along with a main loop implementation for web servers controlled by Guile.

```
(use-modules (web server))
```

The lowest layer is the `<server-impl>` object, which defines a set of hooks to open a server, read a request from a client, write a response to a client, and close a server. These hooks – `open`, `read`, `write`, and `close`, respectively – are bound together in a `<server-impl>` object. Procedures in this module take a `<server-impl>` object, if needed.
A `<server-impl>` may also be looked up by name. If you pass the `http` symbol to `run-server`, Guile looks for a variable named `http` in the `(web server http)` module, which should be bound to a `<server-impl>` object. Such a binding is made by instantiation of the `define-server-impl` syntax. In this way the run-server loop can automatically load other backends if available.

The life cycle of a server goes as follows:

1. The `open` hook is called, to open the server. `open` takes zero or more arguments, depending on the backend, and returns an opaque server socket object, or signals an error.

2. The `read` hook is called, to read a request from a new client. The `read` hook takes one argument, the server socket. It should return three values: an opaque client socket, the request, and the request body. The request should be a `<request>` object, from `(web request)`. The body should be a string or a bytevector, or `#f` if there is no body.
   
   If the read failed, the `read` hook may return `#f` for the client socket, request, and body.

3. A user-provided handler procedure is called, with the request and body as its arguments. The handler should return two values: the response, as a `<response>` record from `(web response)`, and the response body as bytevector, or `#f` if not present.
   
   The response and response body are run through `sanitize-response`, documented below. This allows the handler writer to take some convenient shortcuts: for example, instead of a `<response>`, the handler can simply return an alist of headers, in which case a default response object is constructed with those headers. Instead of a bytevector for the body, the handler can return a string, which will be serialized into an appropriate encoding; or it can return a procedure, which will be called on a port to write out the data. See the `sanitize-response` documentation, for more.

4. The `write` hook is called with three arguments: the client socket, the response, and the body. The `write` hook returns no values.

5. At this point the request handling is complete. For a loop, we loop back and try to read a new request.

6. If the user interrupts the loop, the `close` hook is called on the server socket.

A user may define a server implementation with the following form:

```scheme
define-server-impl name open read write close [Scheme Syntax]
  Make a `<server-impl>` object with the hooks `open`, `read`, `write`, and `close`, and bind it to the symbol `name` in the current module.

lookup-server-impl impl [Scheme Procedure]
  Look up a server implementation. If `impl` is a server implementation already, it is returned directly. If it is a symbol, the binding named `impl` in the `(web server impl)` module is looked up. Otherwise an error is signaled.
  
  Currently a server implementation is a somewhat opaque type, useful only for passing to other procedures in this module, like `read-client`.

The `(web server)` module defines a number of routines that use `<server-impl>` objects to implement parts of a web server. Given that we don’t expose the accessors for the various fields of a `<server-impl>`, indeed these routines are the only procedures with any access to the impl objects.
open-server  impl open-params  [Scheme Procedure]
Open a server for the given implementation. Return one value, the new server object. The implementation’s open procedure is applied to open-params, which should be a list.

read-client  impl server  [Scheme Procedure]
Read a new client from server, by applying the implementation’s read procedure to the server. If successful, return three values: an object corresponding to the client, a request object, and the request body. If any exception occurs, return #f for all three values.

handle-request  handler request body state  [Scheme Procedure]
Handle a given request, returning the response and body.

The response and response body are produced by calling the given handler with request and body as arguments.

The elements of state are also passed to handler as arguments, and may be returned as additional values. The new state, collected from the handler’s return values, is then returned as a list. The idea is that a server loop receives a handler from the user, along with whatever state values the user is interested in, allowing the user’s handler to explicitly manage its state.

sanitize-response  request response body  [Scheme Procedure]
“Sanitize” the given response and body, making them appropriate for the given request.

As a convenience to web handler authors, response may be given as an alist of headers, in which case it is used to construct a default response. Ensures that the response version corresponds to the request version. If body is a string, encodes the string to a bytevector, in an encoding appropriate for response. Adds a content-length and content-type header, as necessary.

If body is a procedure, it is called with a port as an argument, and the output collected as a bytevector. In the future we might try to instead use a compressing, chunk-encoded port, and call this procedure later, in the write-client procedure. Authors are advised not to rely on the procedure being called at any particular time.

write-client  impl server client response body  [Scheme Procedure]
Write an HTTP response and body to client. If the server and client support persistent connections, it is the implementation’s responsibility to keep track of the client thereafter, presumably by attaching it to the server argument somehow.

close-server  impl server  [Scheme Procedure]
Release resources allocated by a previous invocation of open-server.

Given the procedures above, it is a small matter to make a web server:

serve-one-client  handler impl server state  [Scheme Procedure]
Read one request from server, call handler on the request and body, and write the response to the client. Return the new state produced by the handler procedure.
run-server

```scheme
run-server handler [impl='http] [open-params=']([arg ...)]
```

Run Guile’s built-in web server.

*handler* should be a procedure that takes two or more arguments, the HTTP request and request body, and returns two or more values, the response and response body.

For examples, skip ahead to the next section, Section 7.3.10 [Web Examples], page 568.

The response and body will be run through `sanitize-response` before sending back to the client.

Additional arguments to *handler* are taken from *arg ...*. These arguments comprise a *state*. Additional return values are accumulated into a new state, which will be used for subsequent requests. In this way a handler can explicitly manage its state.

The default web server implementation is `http`, which binds to a socket, listening for request on that port.

```
http

```

The default HTTP implementation. We document it as a function with keyword arguments, because that is precisely the way that it is — all of the *open-params* to *run-server* get passed to the implementation’s open function.

```
;; The defaults: localhost:8080
(run-server handler)
;; Same thing
(run-server handler 'http ()
;; On a different port
(run-server handler 'http ('#:port 8081))
;; IPv6
(run-server handler 'http ('#:family AF_INET6 #:port 8081))
;; Custom socket
(run-server handler 'http ('#:socket ,(sudo-make-me-a-socket)))
```

### 7.3.10 Web Examples

Well, enough about the tedious internals. Let’s make a web application!

#### 7.3.10.1 Hello, World!

The first program we have to write, of course, is “Hello, World!”. This means that we have to implement a web handler that does what we want.

Now we define a handler, a function of two arguments and two return values:

```scheme
(define (handler request request-body)
  (values response response-body))
```

In this first example, we take advantage of a short-cut, returning an alist of headers instead of a proper response object. The response body is our payload:

```scheme
(define (hello-world-handler request request-body)
  (values '((content-type . (text/plain))
           "Hello World!"))
```
Now let’s test it, by running a server with this handler. Load up the web server module if you haven’t yet done so, and run a server with this handler:

```
(use-modules (web server))
(run-server hello-world-handler)
```

By default, the web server listens for requests on localhost:8080. Visit that address in your web browser to test. If you see the string, Hello World!, sweet!

### 7.3.10.2 Inspecting the Request

The Hello World program above is a general greeter, responding to all URIs. To make a more exclusive greeter, we need to inspect the request object, and conditionally produce different results. So let’s load up the request, response, and URI modules, and do just that.

```
(use-modules (web server)) ; you probably did this already
(use-modules (web request)
 (web response)
 (web uri))
```

```
(define (request-path-components request)
 (split-and-decode-uri-path (uri-path (request-uri request))))
```

```
(define (hello-hacker-handler request body)
 (if (equal? (request-path-components request)
             '("hacker")
             (values '((content-type . (text/plain)))
                     "Hello hacker!")
             (not-found request)))
```

```
(run-server hello-hacker-handler)
```

Here we see that we have defined a helper to return the components of the URI path as a list of strings, and used that to check for a request to /hacker/. Then the success case is just as before – visit http://localhost:8080/hacker/ in your browser to check.

You should always match against URI path components as decoded by `split-and-decode-uri-path`. The above example will work for /hacker/, //hacker///, and /h%61ck%65r.

But we forgot to define `not-found`! If you are pasting these examples into a REPL, accessing any other URI in your web browser will drop your Guile console into the debugger:

```
<unnamed port>:38:7: In procedure module-lookup:
<unnamed port>:38:7: Unbound variable: not-found
```

Entering a new prompt. Type `,bt` for a backtrace or `,q` to continue. `scheme@(guile-user) [1]>`

So let’s define the function, right there in the debugger. As you probably know, we’ll want to return a 404 response.

```
;; Paste this in your REPL
(define (not-found request)
 (values (build-response #:code 404))
```
(string-append "Resource not found: "
 (uri->string (request-uri request)))))

;; Now paste this to let the web server keep going:
,continue

Now if you access http://localhost/foo/, you get this error message. (Note that some popular web browsers won't show server-generated 404 messages, showing their own instead, unless the 404 message body is long enough.)

7.3.10.3 Higher-Level Interfaces

The web handler interface is a common baseline that all kinds of Guile web applications can use. You will usually want to build something on top of it, however, especially when producing HTML. Here is a simple example that builds up HTML output using SXML (see Section 7.20 [SXML], page 733).

First, load up the modules:

(use-modules (web server)
 (web request)
 (web response)
 (sxml simple))

Now we define a simple templating function that takes a list of HTML body elements, as SXML, and puts them in our super template:

(define (templatize title body)
 '(html (head (title ,title))
 (body ,@body)))

For example, the simplest Hello HTML can be produced like this:

(sxml->xml (templatize "Hello!" '((b "Hi!"))))

⊣

<html><head><title>Hello!</title></head><body><b>Hi!</b></body></html>

Much better to work with Scheme data types than to work with HTML as strings. Now we define a little response helper:

(define* (respond #:optional body #:key
 (status 200)
 (title "Hello hello!")
 (doctype "<!DOCTYPE html>
"
 (content-type-params '((charset . "utf-8")))
 (content-type 'text/html)
 (extra-headers '())
 (sxml (and body (templatize title body))))

(values (build-response
 #:code status
 #:headers '((content-type
 . ((content-type ,@content-type-params))
 ,@extra-headers))

(lambda (port)
 (if sxml
Here we see the power of keyword arguments with default initializers. By the time the arguments are fully parsed, the sxml local variable will hold the templated SXML, ready for sending out to the client.

Also, instead of returning the body as a string, respond gives a procedure, which will be called by the web server to write out the response to the client.

Now, a simple example using this responder, which lays out the incoming headers in an HTML table.

```scheme
(define (debug-page request body)
  (respond
   '((h1 "hello world!"
     (table
      (tr (th "header") (th "value"))
      ,(map (lambda (pair)
             '((tr (td (tt ,(with-output-to-string
                           (lambda () (display (car pair))))))
               (td (tt ,(with-output-to-string
                      (lambda ()
                       (write (cdr pair))))))))
             (request-headers request))))))

  (run-server debug-page)
)
```

Now if you visit any local address in your web browser, we actually see some HTML, finally.

### 7.3.10.4 Conclusion

Well, this is about as far as Guile's built-in web support goes, for now. There are many ways to make a web application, but hopefully by standardizing the most fundamental data types, users will be able to choose the approach that suits them best, while also being able to switch between implementations of the server. This is a relatively new part of Guile, so if you have feedback, let us know, and we can take it into account. Happy hacking on the web!

### 7.4 The (ice-9 getopt-long) Module

The (ice-9 getopt-long) facility is designed to help parse arguments that are passed to Guile programs on the command line, and is modelled after the C library's facility of the same name (see Section “Getopt” in The GNU C Library Reference Manual). For a more low-level interface to command-line argument parsing, See Section 7.5.25 [SRFI-37], page 622.

The (ice-9 getopt-long) module exports two procedures: getopt-long and option-ref.

- **getopt-long** takes a list of strings — the command line arguments — an *option specification*, and some optional keyword parameters. It parses the command line arguments
according to the option specification and keyword parameters, and returns a data structure that encapsulates the results of the parsing.

- `option-ref` then takes the parsed data structure and a specific option’s name, and returns information about that option in particular.

To make these procedures available to your Guile script, include the expression `(use-modules (ice-9 getopt-long))` somewhere near the top, before the first usage of `getopt-long` or `option-ref`.

### 7.4.1 A Short getopt-long Example

This section illustrates how `getopt-long` is used by presenting and dissecting a simple example. The first thing that we need is an option specification that tells `getopt-long` how to parse the command line. This specification is an association list with the long option name as the key. Here is how such a specification might look:

```scheme
(define option-spec
  '((version (single-char #\v) (value #f))
    (help (single-char #\h) (value #f))))
```

This alist tells `getopt-long` that it should accept two long options, called `version` and `help`, and that these options can also be selected by the single-letter abbreviations `v` and `h`, respectively. The `(value #f)` clauses indicate that neither of the options accepts a value.

With this specification we can use `getopt-long` to parse a given command line:

```scheme
(define options (getopt-long (command-line) option-spec))
```

After this call, `options` contains the parsed command line and is ready to be examined by `option-ref`. `option-ref` is called like this:

```scheme
(option-ref options 'help #f)
```

It expects the parsed command line, a symbol indicating the option to examine, and a default value. The default value is returned if the option was not present in the command line, or if the option was present but without a value; otherwise the value from the command line is returned. Usually `option-ref` is called once for each possible option that a script supports.

The following example shows a main program which puts all this together to parse its command line and figure out what the user wanted.

```scheme
(define (main args)
  (let* ((option-spec '((version (single-char #\v) (value #f))
                        (help (single-char #\h) (value #f))))
         (options (getopt-long args option-spec))
         (help-wanted (option-ref options 'help #f))
         (version-wanted (option-ref options 'version #f)))
    (if (or version-wanted help-wanted)
        (begin
          (if version-wanted
              (display "getopt-long-example version 0.3\n")
            (if help-wanted
                (display "\ngetopt-long-example [options]

```
-v, --version Display version
-h, --help Display this help
")

(begin
  (display "Hello, World!") (newline)))))

7.4.2 How to Write an Option Specification

An option specification is an association list (see Section 6.6.20 [Association Lists], page 229) with one list element for each supported option. The key of each list element is a symbol that names the option, while the value is a list of option properties:

```
OPTION-SPEC ::= '( (OPT-NAME1 (PROP-NAME PROP-VALUE) ...) (OPT-NAME2 (PROP-NAME PROP-VALUE) ...) (OPT-NAME3 (PROP-NAME PROP-VALUE) ...) ...)
```

Each `opt-name` specifies the long option name for that option. For example, a list element with `opt-name` `background` specifies an option that can be specified on the command line using the long option `--background`. Further information about the option — whether it takes a value, whether it is required to be present in the command line, and so on — is specified by the option properties.

In the example of the preceding section, we already saw that a long option name can have an equivalent short option character. The equivalent short option character can be set for an option by specifying a `single-char` property in that option’s property list. For example, a list element like `'(output (single-char #\o) ...)` specifies an option with long name `--output` that can also be specified by the equivalent short name `-o`.

The `value` property specifies whether an option requires or accepts a value. If the `value` property is set to `#t`, the option requires a value: `getopt-long` will signal an error if the option name is present without a corresponding value. If set to `#f`, the option does not take a value; in this case, a non-option word that follows the option name in the command line will be treated as a non-option argument. If set to the symbol `optional`, the option accepts a value but does not require one: a non-option word that follows the option name in the command line will be interpreted as that option’s value. If the option name for an option with `'(value optional)` is immediately followed in the command line by another option name, the value for the first option is implicitly `#t`.

The `required?` property indicates whether an option is required to be present in the command line. If the `required?` property is set to `#t`, `getopt-long` will signal an error if the option is not specified.

Finally, the `predicate` property can be used to constrain the possible values of an option. If used, the `predicate` property should be set to a procedure that takes one argument — the proposed option value as a string — and returns either `#t` or `#f` according as the proposed value is or is not acceptable. If the predicate procedure returns `#f`, `getopt-long` will signal an error.

By default, options do not have single-character equivalents, are not required, and do not take values. Where the list element for an option includes a `value` property but no `predicate` property, the option values are unconstrained.
### 7.4.3 Expected Command Line Format

In order for `getopt-long` to correctly parse a command line, that command line must conform to a standard set of rules for how command line options are specified. This section explains what those rules are.

`getopt-long` splits a given command line into several pieces. All elements of the argument list are classified to be either options or normal arguments. Options consist of two dashes and an option name (so-called *long* options), or of one dash followed by a single letter (*short* options).

Options can behave as switches, when they are given without a value, or they can be used to pass a value to the program. The value for an option may be specified using an equals sign, or else is simply the next word in the command line, so the following two invocations are equivalent:

$$
\text{
$ ./foo.scm --output=bar.txt$
$ ./foo.scm --output bar.txt$
}
$$

Short options can be used instead of their long equivalents and can be grouped together after a single dash. For example, the following commands are equivalent.

$$
\text{
$ ./foo.scm --version --help$
$ ./foo.scm -v --help$
$ ./foo.scm -vh$
}
$$

If an option requires a value, it can only be grouped together with other short options if it is the last option in the group; the value is the next argument. So, for example, with the following option specification —

```
((apples (single-char #\a))
 (blimps (single-char #\b) (value #t))
 (catalexis (single-char #\c) (value #t)))
```

— the following command lines would all be acceptable:

$$
\text{
$ ./foo.scm -a -b bang -c couth$
$ ./foo.scm -ab bang -c couth$
$ ./foo.scm -ac couth -b bang$
}
$$

But the next command line is an error, because `-b` is not the last option in its combination, and because a group of short options cannot include two options that both require values:

$$
\text{
$ ./foo.scm -abc couth bang$
}
$$

If an option’s value is optional, `getopt-long` decides whether the option has a value by looking at what follows it in the argument list. If the next element is a string, and it does not appear to be an option itself, then that string is the option’s value.

If the option `--` appears in the argument list, argument parsing stops there and subsequent arguments are returned as ordinary arguments, even if they resemble options. So, with the command line

$$
\text{
$ ./foo.scm --apples "Granny Smith" -- --blimp Goodyear$
}
$$

`getopt-long` will recognize the `--apples` option as having the value "Granny Smith", but will not treat `--blimp` as an option. The strings `--blimp` and `Goodyear` will be returned as ordinary argument strings.
7.4.4 Reference Documentation for getopt-long

getopt-long args grammar [#:stop-at-first-non-option #t]  [Scheme Procedure]

Parse the command line given in args (which must be a list of strings) according to the option specification grammar.

The grammar argument is expected to be a list of this form:

```
((option (property value) ...) ...) 
```

where each option is a symbol denoting the long option, but without the two leading dashes (e.g. version if the option is called --version).

For each option, there may be list of arbitrarily many property/value pairs. The order of the pairs is not important, but every property may only appear once in the property list. The following table lists the possible properties:

- **(single-char char)**
  Accept -char as a single-character equivalent to --option. This is how to specify traditional Unix-style flags.

- **(required? bool)**
  If bool is true, the option is required. getopt-long will raise an error if it is not found in args.

- **(value bool)**
  If bool is #t, the option accepts a value; if it is #f, it does not; and if it is the symbol optional, the option may appear in args with or without a value.

- **(predicate func)**
  If the option accepts a value (i.e. you specified (value #t) for this option), then getopt-long will apply func to the value, and throw an exception if it returns #f. func should be a procedure which accepts a string and returns a boolean value; you may need to use quasiquotes to get it into grammar.

The #:stop-at-first-non-option keyword, if specified with any true value, tells getopt-long to stop when it gets to the first non-option in the command line. That is, at the first word which is neither an option itself, nor the value of an option. Everything in the command line from that word onwards will be returned as non-option arguments.

getopt-long’s args parameter is expected to be a list of strings like the one returned by command-line, with the first element being the name of the command. Therefore getopt-long ignores the first element in args and starts argument interpretation with the second element.

getopt-long signals an error if any of the following conditions hold.
- The option grammar has an invalid syntax.
- One of the options in the argument list was not specified by the grammar.
- A required option is omitted.
- An option which requires an argument did not get one.
• An option that doesn’t accept an argument does get one (this can only happen using the long option --opt=value syntax).
• An option predicate fails.

#:stop-at-first-non-option is useful for command line invocations like guild [--help | --version] [script [script-options]] and cvs [general-options] command [command-options], where there are options at two levels: some generic and understood by the outer command, and some that are specific to the particular script or command being invoked. To use getopt-long in such cases, you would call it twice: firstly with #:stop-at-first-non-option #t, so as to parse any generic options and identify the wanted script or sub-command; secondly, and after trimming off the initial generic command words, with a script- or sub-command-specific option grammar, so as to process those specific options.

7.4.5 Reference Documentation for option-ref

option-ref options key default

Search options for a command line option named key and return its value, if found. If the option has no value, but was given, return #t. If the option was not given, return default. options must be the result of a call to getopt-long.

option-ref always succeeds, either by returning the requested option value from the command line, or the default value.

The special key ‘()' can be used to get a list of all non-option arguments.

7.5 SRFI Support Modules

SRFI is an acronym for Scheme Request For Implementation. The SRFI documents define a lot of syntactic and procedure extensions to standard Scheme as defined in R5RS.

Guile has support for a number of SRFIs. This chapter gives an overview over the available SRFIs and some usage hints. For complete documentation, design rationales and further examples, we advise you to get the relevant SRFI documents from the SRFI home page http://srfi.schemers.org/.

7.5.1 About SRFI Usage

SRFI support in Guile is currently implemented partly in the core library, and partly as add-on modules. That means that some SRFIs are automatically available when the interpreter is started, whereas the other SRFIs require you to use the appropriate support module explicitly.

There are several reasons for this inconsistency. First, the feature checking syntactic form cond-expand (see Section 7.5.2 [SRFI-0], page 577) must be available immediately, because it must be there when the user wants to check for the Scheme implementation, that is, before she can know that it is safe to use use-modules to load SRFI support modules. The second reason is that some features defined in SRFIs had been implemented in Guile before the developers started to add SRFI implementations as modules (for example SRFI-13 (see Section 7.5.11 [SRFI-13], page 603)). In the future, it is possible that SRFIs in the core library might be factored out into separate modules, requiring explicit module loading
when they are needed. So you should be prepared to have to use use-modules someday
in the future to access SRFI-13 bindings. If you want, you can do that already. We have
included the module (srfi srfi-13) in the distribution, which currently does nothing, but
ensures that you can write future-safe code.

Generally, support for a specific SRFI is made available by using modules named (srfi
srfi-number), where number is the number of the SRFI needed. Another possibility
is to use the command line option --use-srfi, which will load the necessary modules
automatically (see Section 4.2 [Invoking Guile], page 35).

7.5.2 SRFI-0 - cond-expand

This SRFI lets a portable Scheme program test for the presence of certain features, and
adapt itself by using different blocks of code, or fail if the necessary features are not available.
There’s no module to load, this is in the Guile core.

A program designed only for Guile will generally not need this mechanism, such a pro-
gram can of course directly use the various documented parts of Guile.

\[
\text{cond-expand (feature body...) ...} \quad \text{[syntax]}
\]

Expand to the body of the first clause whose feature specification is satisfied. It is
an error if no feature is satisfied.

Features are symbols such as srfi-1, and a feature specification can use and, or and
not forms to test combinations. The last clause can be an else, to be used if no
other passes.

For example, define a private version of alist-cons if SRFI-1 is not available.

\[
\begin{align*}
\text{(cond-expand (srfi-1} \\
\text{(else} \\
\text{ (define (alist-cons key val alist) \\
\text{ (cons (cons key val) alist))))}
\end{align*}
\]

Or demand a certain set of SRFIs (list operations, string ports, receive and string
operations), failing if they’re not available.

\[
\text{(cond-expand ((and srfi-1 srfi-6 srfi-8 srfi-13} \\
\text{ )})}
\]

The Guile core has the following features,

- guile
- guile-2 ;; starting from Guile 2.x
- guile-2.2 ;; starting from Guile 2.2
- r5rs
- srfi-0
- srfi-4
- srfi-6
- srfi-13
- srfi-14
- srfi-16
- srfi-23
- srfi-30
Other SRFI feature symbols are defined once their code has been loaded with `use-modules`, since only then are their bindings available.

The `--use-srfi` command line option (see Section 4.2 [Invoking Guile], page 35) is a good way to load SRFIs to satisfy `cond-expand` when running a portable program.

Testing the `guile` feature allows a program to adapt itself to the Guile module system, but still run on other Scheme systems. For example the following demands SRFI-8 (receive), but also knows how to load it with the Guile mechanism.

```scheme
(cond-expand (srfi-8

  (guile
    (use-modules (srfi srfi-8)))))
```

Likewise, testing the `guile-2` feature allows code to be portable between Guile 2.x and previous versions of Guile. For instance, it makes it possible to write code that accounts for Guile 2.x’s compiler, yet be correctly interpreted on 1.8 and earlier versions:

```scheme
(cond-expand (guile-2 (eval-when (compile)

  ;; This must be evaluated at compile time.
  (fluid-set! current-reader my-reader)))

  (guile

  ;; Earlier versions of Guile do not have a
  ;; separate compilation phase.
  (fluid-set! current-reader my-reader)))
```

It should be noted that `cond-expand` is separate from the `*features*` mechanism (see Section 6.23.2 [Feature Tracking], page 452), feature symbols in one are unrelated to those in the other.

7.5.3 SRFI-1 - List library

The list library defined in SRFI-1 contains a lot of useful list processing procedures for construction, examining, destructuring and manipulating lists and pairs.

Since SRFI-1 also defines some procedures which are already contained in R5RS and thus are supported by the Guile core library, some list and pair procedures which appear in the SRFI-1 document may not appear in this section. So when looking for a particular list/pair processing procedure, you should also have a look at the sections Section 6.6.9 [Lists], page 181, and Section 6.6.8 [Pairs], page 178.

7.5.3.1 Constructors

New lists can be constructed by calling one of the following procedures.
xcons d a                   [Scheme Procedure]
   Like cons, but with interchanged arguments. Useful mostly when passed to higher-
   order procedures.

list-tabulate n init-proc   [Scheme Procedure]
   Return an n-element list, where each list element is produced by applying the proce-
   dure init-proc to the corresponding list index. The order in which init-proc is applied
   to the indices is not specified.

list-copy lst               [Scheme Procedure]
   Return a new list containing the elements of the list lst.
   This function differs from the core list-copy (see Section 6.6.9.3 [List Constructors],
   page 182) in accepting improper lists too. And if lst is not a pair at all then it’s treated
   as the final tail of an improper list and simply returned.

circular-list elt1 elt2 . . . [Scheme Procedure]
   Return a circular list containing the given arguments elt1 elt2 . . .

iota count [start step]     [Scheme Procedure]
   Return a list containing count numbers, starting from start and adding step each
   time. The default start is 0, the default step is 1. For example,
   (iota 6) ⇒ (0 1 2 3 4 5)
   (iota 4 2.5 -2) ⇒ (2.5 0.5 -1.5 -3.5)
   This function takes its name from the corresponding primitive in the APL language.

7.5.3.2 Predicates

The procedures in this section test specific properties of lists.

proper-list? obj            [Scheme Procedure]
   Return #t if obj is a proper list, or #f otherwise. This is the same as the core list?
   (see Section 6.6.9.2 [List Predicates], page 181).
   A proper list is a list which ends with the empty list () in the usual way. The empty
   list () itself is a proper list too.
   (proper-list? '(1 2 3)) ⇒ #t
   (proper-list? '()) ⇒ #t

circular-list? obj          [Scheme Procedure]
   Return #t if obj is a circular list, or #f otherwise.
   A circular list is a list where at some point the cdr refers back to a previous pair in
   the list (either the start or some later point), so that following the cdrs takes you
   around in a circle, with no end.
   (define x (list 1 2 3 4))
   (set-cdr! (last-pair x) (cddr x))
   x ⇒ (1 2 3 4 3 4 3 4 . . .)
   (circular-list? x) ⇒ #t
dotted-list? obj

Return #t if obj is a dotted list, or #f otherwise.

A dotted list is a list where the cdr of the last pair is not the empty list (). Any non-pair obj is also considered a dotted list, with length zero.

(dotted-list? '(1 2 . 3)) ⇒ #t
(dotted-list? 99) ⇒ #t

It will be noted that any Scheme object passes exactly one of the above three tests proper-list?, circular-list? and dotted-list?. Non-lists are dotted-list?, finite lists are either proper-list? or dotted-list?, and infinite lists are circular-list?.

null-list? lst

Return #t if lst is the empty list (), #f otherwise. If something else than a proper or circular list is passed as lst, an error is signalled. This procedure is recommended for checking for the end of a list in contexts where dotted lists are not allowed.

not-pair? obj

Return #t is obj is not a pair, #f otherwise. This is shorthand notation (not (pair? obj)) and is supposed to be used for end-of-list checking in contexts where dotted lists are allowed.

list= elt= list1 . . .

Return #t if all argument lists are equal, #f otherwise. List equality is determined by testing whether all lists have the same length and the corresponding elements are equal in the sense of the equality predicate elt=. If no or only one list is given, #t is returned.

7.5.3.3 Selectors

first pair
second pair
third pair
fourth pair
fifth pair
sixth pair
seventh pair
eighth pair
ninth pair
tenth pair

These are synonyms for car, cadr, caddr, . . . .

car+cdr pair

Return two values, the CAR and the CDR of pair.

take lst i
take! lst i

Return a list containing the first i elements of lst.

take! may modify the structure of the argument list lst in order to produce the result.
Chapter 7: Guile Modules

drop \[ \textit{lst} \text{ } \textit{i} \]  
Return a list containing all but the first \( i \) elements of \( \textit{lst} \).

\textbf{take-right} \[ \textit{lst} \text{ } \textit{i} \]  
Return a list containing the \( i \) last elements of \( \textit{lst} \). The return shares a common tail with \( \textit{lst} \).

drop-right \[ \textit{lst} \text{ } \textit{i} \]  
Return a list containing all but the \( i \) last elements of \( \textit{lst} \).

drop-right! \[ \textit{lst} \text{ } \textit{i} \]  
Return a list containing all but the \( i \) last elements of \( \textit{lst} \). \textbf{drop-right} always returns a new list, even when \( i \) is zero. \textbf{drop-right!} may modify the structure of the argument list \( \textit{lst} \) in order to produce the result.

split-at \[ \textit{lst} \text{ } \textit{i} \]  
Return two values, a list containing the first \( i \) elements of the list \( \textit{lst} \) and a list containing the remaining elements.

split-at! \[ \textit{lst} \text{ } \textit{i} \]  
Return two values, a list containing the first \( i \) elements of the list \( \textit{lst} \) and a list containing the remaining elements. \textbf{split-at!} may modify the structure of the argument list \( \textit{lst} \) in order to produce the result.

last \[ \textit{lst} \]  
Return the last element of the non-empty, finite list \( \textit{lst} \).

7.5.3.4 Length, Append, Concatenate, etc.

\textbf{length+} \[ \textit{lst} \]  
Return the length of the argument list \( \textit{lst} \). When \( \textit{lst} \) is a circular list, \#f is returned.

concatenate \[ \textit{list-of-lists} \]  
Construct a list by appending all lists in \( \textit{list-of-lists} \).

concatenate! \[ \textit{list-of-lists} \]  
Construct a list by appending all lists in \( \textit{list-of-lists} \). \textbf{concatenate!} may modify the structure of the given lists in order to produce the result.

concatenate is the same as \( \textbf{apply append \textit{list-of-lists}} \). It exists because some Scheme implementations have a limit on the number of arguments a function takes, which the \textbf{apply} might exceed. In Guile there is no such limit.

append-reverse \[ \textit{rev-head} \text{ } \textit{tail} \]  
Reverse \( \textit{rev-head} \), append \( \textit{tail} \) to it, and return the result. This is equivalent to \( \textbf{append (reverse \textit{rev-head}) \textit{tail}} \), but its implementation is more efficient.

\( \textbf{append-reverse '}(1 \ 2 \ 3) \ '}(4 \ 5 \ 6) \Rightarrow (3 \ 2 \ 1 \ 4 \ 5 \ 6) \)

append-reverse! may modify \( \textit{rev-head} \) in order to produce the result.

\textbf{zip} \[ \textit{lst1} \text{ } \textit{lst2} \ldots \]  
Return a list as long as the shortest of the argument lists, where each element is a list. The first list contains the first elements of the argument lists, the second list contains the second elements, and so on.
unzip1 lst  
unzip2 lst  
unzip3 lst  
unzip4 lst  
unzip5 lst

unzip1 takes a list of lists, and returns a list containing the first elements of each list, unzip2 returns two lists, the first containing the first elements of each lists and the second containing the second elements of each lists, and so on.

count pred lst1 lst2 ...

Return a count of the number of times pred returns true when called on elements from the given lists.
pred is called with N parameters (pred elem1 ... elemN ), each element being from the corresponding list. The first call is with the first element of each list, the second with the second element from each, and so on.

Counting stops when the end of the shortest list is reached. At least one list must be non-circular.

7.5.3.5 Fold, Unfold & Map

fold proc init lst1 lst2 ...
fold-right proc init lst1 lst2 ...

Apply proc to the elements of lst1 lst2 ... to build a result, and return that result.

Each proc call is (proc elem1 elem2 ... previous), where elem1 is from lst1, elem2 is from lst2, and so on. previous is the return from the previous call to proc, or the given init for the first call. If any list is empty, just init is returned.

fold works through the list elements from first to last. The following shows a list reversal and the calls it makes,

(fold cons () '(1 2 3))

(cons 1 '())
(cons 2 '(1))
(cons 3 '(2 1))
⇒ (3 2 1)

fold-right works through the list elements from last to first, ie. from the right. So for example the following finds the longest string, and the last among equal longest,

(fold-right (lambda (str prev)
  (if (> (string-length str) (string-length prev))
    str
    prev))

  ""

  '("x" "abc" "xyz" "jk")
⇒ "xyz"

If lst1 lst2 ... have different lengths, fold stops when the end of the shortest is reached; fold-right commences at the last element of the shortest. Ie. elements
past the length of the shortest are ignored in the other \textit{lst}s. At least one \textit{lst} must be non-circular.

\texttt{fold} should be preferred over \texttt{fold-right} if the order of processing doesn’t matter, or can be arranged either way, since \texttt{fold} is a little more efficient.

The way \texttt{fold} builds a result from iterating is quite general, it can do more than other iterations like say \texttt{map} or \texttt{filter}. The following for example removes adjacent duplicate elements from a list,

\begin{verbatim}
(define (delete-adjacent-duplicates lst)
  (fold-right (lambda (elem ret)
                  (if (equal? elem (first ret))
                      ret
                      (cons elem ret)))
              (list (last lst))
              lst))
\end{verbatim}

\begin{verbatim}
(\texttt{delete-adjacent-duplicates \textquoteleft1 2 3 3 4 4 4 5\textquoteright})
⇒ \textquoteleft1 2 3 4 5\textquoteright
\end{verbatim}

Clearly the same sort of thing can be done with a \texttt{for-each} and a variable in which to build the result, but a self-contained \texttt{proc} can be re-used in multiple contexts, where a \texttt{for-each} would have to be written out each time.

\begin{verbatim}
pair-fold proc init lst1 lst2 ... [Scheme Procedure]  
pair-fold-right proc init lst1 lst2 ... [Scheme Procedure] 
  The same as \texttt{fold} and \texttt{fold-right}, but apply \texttt{proc} to the pairs of the lists instead of the list elements.
\end{verbatim}

\begin{verbatim}
reduce proc default lst     [Scheme Procedure]  
reduce-right proc default lst [Scheme Procedure]  
  \texttt{reduce} is a variant of \texttt{fold}, where the first call to \texttt{proc} is on two elements from \texttt{lst}, rather than one element and a given initial value.
  
  If \texttt{lst} is empty, \texttt{reduce} returns \texttt{default} (this is the only use for \texttt{default}). If \texttt{lst} has just one element then that’s the return value. Otherwise \texttt{proc} is called on the elements of \texttt{lst}.
  
  Each \texttt{proc} call is \texttt{(proc elem previous)}, where \texttt{elem} is from \texttt{lst} (the second and subsequent elements of \texttt{lst}), and \texttt{previous} is the return from the previous call to \texttt{proc}. The first element of \texttt{lst} is the \texttt{previous} for the first call to \texttt{proc}.
  
  For example, the following adds a list of numbers, the calls made to + are shown. (Of course + accepts multiple arguments and can add a list directly, with \texttt{apply}.)
\end{verbatim}

\begin{verbatim}
(reduce + 0 \textquoteleft5 6 7\textquoteright) ⇒ 18
\end{verbatim}

\begin{verbatim}
(+ 6 5) ⇒ 11
(+ 7 11) ⇒ 18
\end{verbatim}

\texttt{reduce} can be used instead of \texttt{fold} where the \texttt{init} value is an “identity”, meaning a value which under \texttt{proc} doesn’t change the result, in this case 0 is an identity since (+ 5 0) is just 5. \texttt{reduce} avoids that unnecessary call.
reduce-right is a similar variation on fold-right, working from the end (ie. the right) of lst. The last element of lst is the previous for the first call to proc, and the elem values go from the second last.

reduce should be preferred over reduce-right if the order of processing doesn’t matter, or can be arranged either way, since reduce is a little more efficient.

unfold p f g seed [tail-gen] [Scheme Procedure]

unfold is defined as follows:

\[(\text{unfold } p \ f \ g \ \text{seed}) = \]
\[
\begin{cases} 
   (\text{if } (p \ \text{seed}) \ (\text{tail-gen } \text{seed}) & \\
   (\text{cons } (f \ \text{seed}) & \\
   (\text{unfold } p \ f \ g \ (g \ \text{seed}))) & 
\end{cases}
\]

\(p\) Determines when to stop unfolding.

\(f\) Maps each seed value to the corresponding list element.

\(g\) Maps each seed value to next seed value.

\(seed\) The state value for the unfold.

\(tail-gen\) Creates the tail of the list; defaults to \((\lambda x. '())\).

\(g\) produces a series of seed values, which are mapped to list elements by \(f\). These elements are put into a list in left-to-right order, and \(p\) tells when to stop unfolding.

unfold-right p f g seed [tail] [Scheme Procedure]

Construct a list with the following loop.

\[(\text{let lp } ((\text{seed seed}) \ (\text{lis tail})) \]
\[
\begin{cases} 
   (\text{if } (p \ \text{seed}) \ \text{lis} & \\
   (\text{lp } (g \ \text{seed}) & \\
   (\text{cons } (f \ \text{seed}) \ \text{lis}))) & 
\end{cases}
\]

\(p\) Determines when to stop unfolding.

\(f\) Maps each seed value to the corresponding list element.

\(g\) Maps each seed value to next seed value.

\(seed\) The state value for the unfold.

\(tail\) The tail of the list; defaults to \('()\).

map f lst1 lst2 ... [Scheme Procedure]

Map the procedure over the list(s) \(lst1, lst2, ...\) and return a list containing the results of the procedure applications. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The result list will have the same length as the shortest argument lists. The order in which \(f\) will be applied to the list element(s) is not specified.

for-each f lst1 lst2 ... [Scheme Procedure]

Apply the procedure \(f\) to each pair of corresponding elements of the list(s) \(lst1, lst2, ...\). The return value is not specified. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The shortest argument list determines the number of times \(f\) is called. \(f\) will be applied to the list elements in left-to-right order.
**Scheme Procedure**

**append-map** \( f \ lst1 \ lst2 \ldots \)

Equivalent to

\[
(\text{apply} \ \text{append} \ (\text{map} \ f \ \text{clist1} \ \text{clist2} \ \ldots))
\]

and

\[
(\text{apply} \ \text{append!} \ (\text{map} \ f \ \text{clist1} \ \text{clist2} \ \ldots))
\]

Map \( f \) over the elements of the lists, just as in the \text{map} function. However, the results of the applications are appended together to make the final result. \text{append-map} uses \text{append} to append the results together; \text{append-map!} uses \text{append!}.

The dynamic order in which the various applications of \( f \) are made is not specified.

**Scheme Procedure**

**map!** \( f \ lst1 \ lst2 \ldots \)

Linear-update variant of \text{map} – \text{map!} is allowed, but not required, to alter the cons cells of \text{lst1} to construct the result list.

The dynamic order in which the various applications of \( f \) are made is not specified. In the n-ary case, \text{lst2}, \text{lst3}, \ldots must have at least as many elements as \text{lst1}.

**Scheme Procedure**

**pair-for-each** \( f \ lst1 \ lst2 \ldots \)

Like \text{for-each}, but applies the procedure \( f \) to the pairs from which the argument lists are constructed, instead of the list elements. The return value is not specified.

**Scheme Procedure**

**filter-map** \( f \ lst1 \ lst2 \ldots \)

Like \text{map}, but only results from the applications of \( f \) which are true are saved in the result list.

### 7.5.3.6 Filtering and Partitioning

Filtering means to collect all elements from a list which satisfy a specific condition. Partitioning a list means to make two groups of list elements, one which contains the elements satisfying a condition, and the other for the elements which don’t.

The \text{filter} and \text{filter!} functions are implemented in the Guile core, See Section 6.6.9.6 [List Modification], page 184.

**Scheme Procedure**

**partition** \( \text{pred} \ lst \)

**Scheme Procedure**

**partition!** \( \text{pred} \ lst \)

Split \( lst \) into those elements which do and don’t satisfy the predicate \( \text{pred} \).

The return is two values (see Section 6.13.7 [Multiple Values], page 309), the first being a list of all elements from \( lst \) which satisfy \( \text{pred} \), the second a list of those which do not.

The elements in the result lists are in the same order as in \( lst \) but the order in which the calls \((\text{pred elem})\) are made on the list elements is unspecified.

\text{partition} does not change \( lst \), but one of the returned lists may share a tail with it. \text{partition!} may modify \( lst \) to construct its return.

**Scheme Procedure**

**remove** \( \text{pred} \ lst \)

**Scheme Procedure**

**remove!** \( \text{pred} \ lst \)

Return a list containing all elements from \( lst \) which do not satisfy the predicate \( \text{pred} \).

The elements in the result list have the same order as in \( lst \). The order in which \( \text{pred} \) is applied to the list elements is not specified.
remove! is allowed, but not required to modify the structure of the input list.

### 7.5.3.7 Searching

The procedures for searching elements in lists either accept a predicate or a comparison object for determining which elements are to be searched.

**find** `pred lst`  
Returns the first element of `lst` which satisfies the predicate `pred` and `#f` if no such element is found.

**find-tail** `pred lst`  
Returns the first pair of `lst` whose car satisfies the predicate `pred` and `#f` if no such element is found.

**take-while** `pred lst`  
**take-while!** `pred lst`  
Returns the longest initial prefix of `lst` whose elements all satisfy the predicate `pred`. `take-while!` is allowed, but not required to modify the input list while producing the result.

**drop-while** `pred lst`  
Drop the longest initial prefix of `lst` whose elements all satisfy the predicate `pred`.

**span** `pred lst`  
**span!** `pred lst`  
**break** `pred lst`  
**break!** `pred lst`  
span splits the list `lst` into the longest initial prefix whose elements all satisfy the predicate `pred`, and the remaining tail. `break` inverts the sense of the predicate. `span!` and `break!` are allowed, but not required to modify the structure of the input list `lst` in order to produce the result.

Note that the name `break` conflicts with the `break` binding established by `while` (see Section 6.13.4 [while do], page 301). Applications wanting to use `break` from within a `while` loop will need to make a new define under a different name.

**any** `pred lst1 lst2 ...`  
Test whether any set of elements from `lst1 lst2 ...` satisfies `pred`. If so, the return value is the return value from the successful `pred` call! or if not, the return value is `#f`.

If there are `n` list arguments, then `pred` must be a predicate taking `n` arguments. Each `pred` call is `(pred elem1 elem2 ...)` taking an element from each `lst`. The calls are made successively for the first, second, etc. elements of the lists, stopping when `pred` returns non-`#f`, or when the end of the shortest list is reached.

The `pred` call on the last set of elements (i.e., when the end of the shortest list has been reached), if that point is reached, is a tail call.
every pred lst1 lst2 ...

Test whether every set of elements from lst1 lst2 ... satisfies pred. If so, the return value is the return from the final pred call, or if not, the return value is #f.

If there are n list arguments, then pred must be a predicate taking n arguments. Each pred call is (pred elem1 elem2 ...) taking an element from each lst. The calls are made successively for the first, second, etc. elements of the lists, stopping if pred returns #f, or when the end of any of the lists is reached.

The pred call on the last set of elements (i.e., when the end of the shortest list has been reached) is a tail call.

If one of lst1 lst2 ... is empty then no calls to pred are made, and the return value is #t.

list-index pred lst1 lst2 ...

Return the index of the first set of elements, one from each of lst1 lst2 ..., which satisfies pred.

pred is called as (elem1 elem2 ...). Searching stops when the end of the shortest lst is reached. The return index starts from 0 for the first set of elements. If no set of elements pass, then the return value is #f.

(list-index odd? '(2 4 6 9)) ⇒ 3
(list-index = '(1 2 3) '(3 1 2)) ⇒ #f

member x lst [=]

Return the first sublist of lst whose car is equal to x. If x does not appear in lst, return #f.

Equality is determined by equal?, or by the equality predicate = if given. = is called (= x elem), ie. with the given x first, so for example to find the first element greater than 5,

(member 5 '(3 5 1 7 2 9) <) ⇒ (7 2 9)

This version of member extends the core member (see Section 6.6.9.7 [List Searching], page 185) by accepting an equality predicate.

7.5.3.8 Deleting

delete x lst [=]
delete! x lst [=]

Return a list containing the elements of lst but with those equal to x deleted. The returned elements will be in the same order as they were in lst.

Equality is determined by the = predicate, or equal? if not given. An equality call is made just once for each element, but the order in which the calls are made on the elements is unspecified.

The equality calls are always (= x elem), ie. the given x is first. This means for instance elements greater than 5 can be deleted with (delete 5 lst <).

delete does not modify lst, but the return might share a common tail with lst.
delete! may modify the structure of lst to construct its return.

These functions extend the core delete and delete! (see Section 6.6.9.6 [List Modification], page 184) in accepting an equality predicate. See also lset-difference (see
Section 7.5.3.10 [SRFI-1 Set Operations], page 589) for deleting multiple elements from a list.

`delete-duplicates lst [=]` [Scheme Procedure]
`delete-duplicates! lst [=]` [Scheme Procedure]
Return a list containing the elements of `lst` but without duplicates.

When elements are equal, only the first in `lst` is retained. Equal elements can be anywhere in `lst`, they don’t have to be adjacent. The returned list will have the retained elements in the same order as they were in `lst`.

Equality is determined by the `=` predicate, or `equal?` if not given. Calls `(``=` `x` `y`)` are made with element `x` being before `y` in `lst`. A call is made at most once for each combination, but the sequence of the calls across the elements is unspecified.

`delete-duplicates` does not modify `lst`, but the return might share a common tail with `lst`. `delete-duplicates!` may modify the structure of `lst` to construct its return.

In the worst case, this is an `O(N^2)` algorithm because it must check each element against all those preceding it. For long lists it is more efficient to sort and then compare only adjacent elements.

### 7.5.3.9 Association Lists

Association lists are described in detail in section Section 6.6.20 [Association Lists], page 229. The present section only documents the additional procedures for dealing with association lists defined by SRFI-1.

`assoc key alist [=]` [Scheme Procedure]
Return the pair from `alist` which matches `key`. This extends the core `assoc` (see Section 6.6.20.3 [Retrieving Alist Entries], page 232) by taking an optional `=` comparison procedure.

The default comparison is `equal?`. If an `=` parameter is given it’s called `(``=` `key` `alist` `car`)`, i.e. the given target `key` is the first argument, and a `car` from `alist` is second.

For example a case-insensitive string lookup,

```
(assoc "yy" '(("XX" . 1) ("YY" . 2)) string-ci=?)
⇒ ("YY" . 2)
```

`alist-cons key datum alist` [Scheme Procedure]
Cons a new association `key` and `datum` onto `alist` and return the result. This is equivalent to

```
(cons (cons key datum) alist)
```

`acons` (see Section 6.6.20.2 [Adding or Setting Alist Entries], page 230) in the Guile core does the same thing.

`alist-copy alist` [Scheme Procedure]
Return a newly allocated copy of `alist`, that means that the spine of the list as well as the pairs are copied.
alist-delete  key  alist  [=]  [Scheme Procedure]
alist-delete!  key  alist  [=]  [Scheme Procedure]

Return a list containing the elements of alist but with those elements whose keys are equal to key deleted. The returned elements will be in the same order as they were in alist.

Equality is determined by the = predicate, or equal? if not given. The order in which elements are tested is unspecified, but each equality call is made (= key alist key), i.e. the given key parameter is first and the key from alist second. This means for instance all associations with a key greater than 5 can be removed with (alist-delete 5 alist <).

alist-delete does not modify alist, but the return might share a common tail with alist. alist-delete! may modify the list structure of alist to construct its return.

7.5.3.10 Set Operations on Lists

Lists can be used to represent sets of objects. The procedures in this section operate on such lists as sets.

Note that lists are not an efficient way to implement large sets. The procedures here typically take time \( m \times n \) when operating on \( m \) and \( n \) element lists. Other data structures like trees, bitsets (see Section 6.6.11 [Bit Vectors], page 191) or hash tables (see Section 6.6.22 [Hash Tables], page 237) are faster.

All these procedures take an equality predicate as the first argument. This predicate is used for testing the objects in the list sets for sameness. This predicate must be consistent with eq? (see Section 6.11.1 [Equality], page 283) in the sense that if two list elements are eq? then they must also be equal under the predicate. This simply means a given object must be equal to itself.

lset<=  =  list  . . .  [Scheme Procedure]

Return \#t if each list is a subset of the one following it. I.e., list1 is a subset of list2, list2 is a subset of list3, etc., for as many lists as given. If only one list or no lists are given, the return value is \#t.

A list \( x \) is a subset of \( y \) if each element of \( x \) is equal to some element in \( y \). Elements are compared using the given = procedure, called as (= xelem yelem).

\[
\begin{align*}
lset<=  &  \Rightarrow  \#t \\
lset<=  &  \Rightarrow  \#f \\
lset<=  &  \Rightarrow  \#t
\end{align*}
\]

lset=  =  list  . . .  [Scheme Procedure]

Return \#t if all argument lists are set-equal. list1 is compared to list2, list2 to list3, etc., for as many lists as given. If only one list or no lists are given, the return value is \#t.

Two lists \( x \) and \( y \) are set-equal if each element of \( x \) is equal to some element of \( y \) and conversely each element of \( y \) is equal to some element of \( x \). The order of the elements in the lists doesn’t matter. Element equality is determined with the given = procedure, called as (= xelem yelem), but exactly which calls are made is unspecified.

\[
\begin{align*}
lset=  &  \Rightarrow  \#t \\
lset=  &  \Rightarrow  \#t
\end{align*}
\]
(lset= string-ci=? '("a" "A" "b") '("B" "b" "a")) ⇒ #t

lset-adjoin = list elem ...        [Scheme Procedure]
           Add to list any of the given elems not already in the list. elems are coned onto the
           start of list (so the return value shares a common tail with list), but the order that
           the elems are added is unspecified.
           The given = procedure is used for comparing elements, called as (= listelem elem),
           i.e., the second argument is one of the given elem parameters.
           (lset-adjoin eqv? '(1 2 3) 4 1 5) ⇒ (5 4 1 2 3)

lset-union = list ...        [Scheme Procedure]
lset-union! = list ...        [Scheme Procedure]
           Return the union of the argument list sets. The result is built by taking the union of
           list1 and list2, then the union of that with list3, etc., for as many lists as given. For
           one list argument that list itself is the result, for no list arguments the result is the
           empty list.
           The union of two lists x and y is formed as follows. If x is empty then the result is y.
           Otherwise start with x as the result and consider each y element (from first to last).
           A y element not equal to something already in the result is coned onto the result.
           The given = procedure is used for comparing elements, called as (= relem yelem).
           The first argument is from the result accumulated so far, and the second is from the
           list being union-ed in. But exactly which calls are made is otherwise unspecified.
           Notice that duplicate elements in list1 (or the first non-empty list) are preserved, but
           that repeated elements in subsequent lists are only added once.
           (lset-union eqv?) ⇒ ()
           (lset-union eqv? '(1 2 3)) ⇒ (1 2 3)
           (lset-union eqv? '(1 2 1 3) '(2 4 5) '(5)) ⇒ (5 4 1 2 1 3)

lset-union doesn't change the given lists but the result may share a tail with the
first non-empty list. lset-union! can modify all of the given lists to form the result.

lset-intersection = list1 list2 ...        [Scheme Procedure]
lset-intersection! = list1 list2 ...        [Scheme Procedure]
           Return the intersection of list1 with the other argument lists, meaning those elements
           of list1 which are also in all of list2 etc. For one list argument, just that list is returned.
           The test for an element of list1 to be in the return is simply that it's equal to some
           element in each of list2 etc. Notice this means an element appearing twice in list1
           but only once in each of list2 etc will go into the return twice. The return has its
           elements in the same order as they were in list1.
           The given = procedure is used for comparing elements, called as (= elem1 elemN).
           The first argument is from list1 and the second is from one of the subsequent lists.
           But exactly which calls are made and in what order is unspecified.
           (lset-intersection eqv? '(x y)) ⇒ (x y)
           (lset-intersection eqv? '(1 2 3) '(4 3 2)) ⇒ (2 3)
           (lset-intersection eqv? '(1 1 2 2) '(1 2) '(2 1) '(2)) ⇒ (2 2)
           The return from lset-intersection may share a tail with list1.
           lset-intersection! may modify list1 to form its result.
lset-difference = list1 list2 ...  [Scheme Procedure]
lset-difference! = list1 list2 ...  [Scheme Procedure]

Return list1 with any elements in list2, list3 etc removed (ie. subtracted). For one list argument, just that list is returned.

The given = procedure is used for comparing elements, called as (= elem1 elemN). The first argument is from list1 and the second from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

  (lset-difference eqv? '(x y)) ⇒ (x y)
  (lset-difference eqv? '(1 2 3) '(3 1)) ⇒ (2)
  (lset-difference eqv? '(1 2 3) '(3) '(2)) ⇒ (1)

The return from lset-difference may share a tail with list1. lset-difference! may modify list1 to form its result.

lset-diff+intersection = list1 list2 ...  [Scheme Procedure]
lset-diff+intersection! = list1 list2 ...  [Scheme Procedure]

Return two values (see Section 6.13.7 [Multiple Values], page 309), the difference and intersection of the argument lists as per lset-difference and lset-intersection above.

For two list arguments this partitions list1 into those elements of list1 which are in list2 and not in list2. (But for more than two arguments there can be elements of list1 which are neither part of the difference nor the intersection.)

One of the return values from lset-diff+intersection may share a tail with list1. lset-diff+intersection! may modify list1 to form its results.

lset-xor = list ...  [Scheme Procedure]
lset-xor! = list ...  [Scheme Procedure]

Return an XOR of the argument lists. For two lists this means those elements which are in exactly one of the lists. For more than two lists it means those elements which appear in an odd number of the lists.

To be precise, the XOR of two lists x and y is formed by taking those elements of x not equal to any element of y, plus those elements of y not equal to any element of x. Equality is determined with the given = procedure, called as (= e1 e2). One argument is from x and the other from y, but which way around is unspecified. Exactly which calls are made is also unspecified, as is the order of the elements in the result.

  (lset-xor eqv? '(x y)) ⇒ (x y)
  (lset-xor eqv? '(1 2 3) '(4 3 2)) ⇒ (4 1)

The return from lset-xor may share a tail with one of the list arguments. lset-xor! may modify list1 to form its result.

7.5.4 SRFI-2 - and-let*

The following syntax can be obtained with

  (use-modules (srfi srfi-2))

or alternatively

  (use-modules (ice-9 and-let-star))
and-let* (clause ...) body ...

A combination of and and let*.

Each clause is evaluated in turn, and if #f is obtained then evaluation stops and #f is returned. If all are non-#f then body is evaluated and the last form gives the return value, or if body is empty then the result is #t. Each clause should be one of the following,

(symbol expr)
Evaluate expr, check for #f, and bind it to symbol. Like let*, that binding is available to subsequent clauses.

(expr)  Evaluate expr and check for #f.

symbol    Get the value bound to symbol and check for #f.

Notice that (expr) has an “extra” pair of parentheses, for instance ((eq? x y)). One way to remember this is to imagine the symbol in (symbol expr) is omitted.

and-let* is good for calculations where a #f value means termination, but where a non-#f value is going to be needed in subsequent expressions.

The following illustrates this, it returns text between brackets ‘[...]’ in a string, or #f if there are no such brackets (ie. either string-index gives #f).

(define (extract-brackets str)
  (and-let* ((start (string-index str #\[))
    (end (string-index str #\] start)))
    (substring str (+ start end))))

The following shows plain variables and expressions tested too. diagnostic-levels is taken to be an alist associating a diagnostic type with a level. str is printed only if the type is known and its level is high enough.

(define (show-diagnostic type str)
  (and-let* (want-diagnostics
    (level (assq-ref diagnostic-levels type))
    ((>= level current-diagnostic-level)))
    (display str)))

The advantage of and-let* is that an extended sequence of expressions and tests doesn’t require lots of nesting as would arise from separate and and let*, or from cond with =>.

7.5.5 SRFI-4 - Homogeneous numeric vector datatypes

SRFI-4 provides an interface to uniform numeric vectors: vectors whose elements are all of a single numeric type. Guile offers uniform numeric vectors for signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers, two sizes of floating point values, and, as an extension to SRFI-4, complex floating-point numbers of these two sizes.

The standard SRFI-4 procedures and data types may be included via loading the appropriate module:

(use-modules (srfi srfi-4))

This module is currently a part of the default Guile environment, but it is a good practice to explicitly import the module. In the future, using SRFI-4 procedures without importing
the SRFI-4 module will cause a deprecation message to be printed. (Of course, one may call the C functions at any time. Would that C had modules!)

### 7.5.5.1 SRFI-4 - Overview

Uniform numeric vectors can be useful since they consume less memory than the non-uniform, general vectors. Also, since the types they can store correspond directly to C types, it is easier to work with them efficiently on a low level. Consider image processing as an example, where you want to apply a filter to some image. While you could store the pixels of an image in a general vector and write a general convolution function, things are much more efficient with uniform vectors: the convolution function knows that all pixels are unsigned 8-bit values (say), and can use a very tight inner loop.

This is implemented in Scheme by having the compiler notice calls to the SRFI-4 accessors, and inline them to appropriate compiled code. From C you have access to the raw array; functions for efficiently working with uniform numeric vectors from C are listed at the end of this section.

Uniform numeric vectors are the special case of one dimensional uniform numeric arrays.

There are 12 standard kinds of uniform numeric vectors, and they all have their own complement of constructors, accessors, and so on. Procedures that operate on a specific kind of uniform numeric vector have a “tag” in their name, indicating the element type.

- `u8` unsigned 8-bit integers
- `s8` signed 8-bit integers
- `u16` unsigned 16-bit integers
- `s16` signed 16-bit integers
- `u32` unsigned 32-bit integers
- `s32` signed 32-bit integers
- `u64` unsigned 64-bit integers
- `s64` signed 64-bit integers
- `f32` the C type `float`
- `f64` the C type `double`

In addition, Guile supports uniform arrays of complex numbers, with the nonstandard tags:

- `c32` complex numbers in rectangular form with the real and imaginary part being a `float`
- `c64` complex numbers in rectangular form with the real and imaginary part being a `double`

The external representation (ie. read syntax) for these vectors is similar to normal Scheme vectors, but with an additional tag from the tables above indicating the vector’s type. For example,

```
#u16(1 2 3)
#f64(3.1415 2.71)
```
Note that the read syntax for floating-point here conflicts with #f for false. In Standard Scheme one can write (1 #f3) for a three element list (1 #f 3), but for Guile (1 #f3) is invalid. (1 #f 3) is almost certainly what one should write anyway to make the intention clear, so this is rarely a problem.

### 7.5.5.2 SRFI-4 - API

Note that the c32 and c64 functions are only available from (srfi srfi-4 gnu).

```
(u8vector? obj)  ; [Scheme Procedure]
s8vector? obj    ; [Scheme Procedure]
u16vector? obj   ; [Scheme Procedure]
s16vector? obj   ; [Scheme Procedure]
u32vector? obj   ; [Scheme Procedure]
s32vector? obj   ; [Scheme Procedure]
u64vector? obj   ; [Scheme Procedure]
s64vector? obj   ; [Scheme Procedure]
f32vector? obj   ; [Scheme Procedure]
f64vector? obj   ; [Scheme Procedure]
c32vector? obj   ; [Scheme Procedure]
c64vector? obj   ; [Scheme Procedure]

(scm_u8vector_p obj)  ; [C Function]
(scm_s8vector_p obj)   ; [C Function]
(scm_u16vector_p obj)  ; [C Function]
(scm_s16vector_p obj)  ; [C Function]
(scm_u32vector_p obj)  ; [C Function]
(scm_s32vector_p obj)  ; [C Function]
(scm_u64vector_p obj)  ; [C Function]
(scm_s64vector_p obj)  ; [C Function]
(scm_f32vector_p obj)  ; [C Function]
(scm_f64vector_p obj)  ; [C Function]
(scm_c32vector_p obj)  ; [C Function]
(scm_c64vector_p obj)  ; [C Function]
```

Return #t if obj is a homogeneous numeric vector of the indicated type.

```
(make-u8vector n [value])        ; [Scheme Procedure]
(make-s8vector n [value])        ; [Scheme Procedure]
(make-u16vector n [value])       ; [Scheme Procedure]
(make-s16vector n [value])       ; [Scheme Procedure]
(make-u32vector n [value])       ; [Scheme Procedure]
(make-s32vector n [value])       ; [Scheme Procedure]
(make-u64vector n [value])       ; [Scheme Procedure]
(make-s64vector n [value])       ; [Scheme Procedure]
(make-f32vector n [value])       ; [Scheme Procedure]
(make-f64vector n [value])       ; [Scheme Procedure]
(make-c32vector n [value])       ; [Scheme Procedure]
(make-c64vector n [value])       ; [Scheme Procedure]
(scm_make_u8vector (n, value))   ; [C Function]
```
Return a newly allocated homogeneous numeric vector holding \( n \) elements of the indicated type. If \( \text{value} \) is given, the vector is initialized with that value, otherwise the contents are unspecified.

Return a newly allocated homogeneous numeric vector of the indicated type, holding the given parameter \( \text{values} \). The vector length is the number of parameters given.
Return the number of elements in vec.

Return the element at index i in vec. The first element in vec is index 0.
Set the element at index \( i \) in \( \text{vec} \) to \( \text{value} \). The first element in \( \text{vec} \) is index 0. The return value is unspecified.
\textbf{scm\_c32vector\_to\_list} (\textit{vec}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_c64vector\_to\_list} (\textit{vec}) \hspace{1cm} \textbf{[C Function]}

Return a newly allocated list holding all elements of \textit{vec}.

\textbf{list\->u8vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->s8vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->u16vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->s16vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->u32vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->s32vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->u64vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->s64vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->f32vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->f64vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->c32vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}
\textbf{list\->c64vector} \textit{lst} \hspace{1cm} \textbf{[Scheme Procedure]}

\textbf{scm\_list\_to\_u8vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_s8vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_u16vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_s16vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_u32vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_s32vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_u64vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_s64vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_f32vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_f64vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_c32vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}
\textbf{scm\_list\_to\_c64vector} (\textit{lst}) \hspace{1cm} \textbf{[C Function]}

Return a newly allocated homogeneous numeric vector of the indicated type, initialized with the elements of the list \textit{lst}.

\textbf{SCM} \textbf{scm\_take\_u8vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_s8vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_u16vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_s16vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_u32vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_s32vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_u64vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_s64vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_f32vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_f64vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_c32vector} \hspace{1cm} \textbf{[C Function]}
\textbf{SCM} \textbf{scm\_take\_c64vector} \hspace{1cm} \textbf{[C Function]}

Return a new uniform numeric vector of the indicated type and length that uses the memory pointed to by \textit{data} to store its elements. This memory will eventually be freed with \texttt{free}. The argument \textit{len} specifies the number of elements in \textit{data}, not its size in bytes.
The c32 and c64 variants take a pointer to a C array of `floats` or `doubles`. The real parts of the complex numbers are at even indices in that array, the corresponding imaginary parts are at the following odd index.

```c
const scm_t_uint8 * scm_u8vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_int8 * scm_s8vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_uint16 * scm_u16vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_int16 * scm_s16vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_uint32 * scm_u32vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_int32 * scm_s32vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_uint64 * scm_u64vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const scm_t_int64 * scm_s64vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const float * scm_f32vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const double * scm_f64vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const float * scm_c32vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
const double * scm_c64vector_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
```

Like `scm_vector_elements` (see Section 6.6.10.4 [Vector Accessing from C], page 189), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

```c
scm_t_uint8 * scm_u8vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_int8 * scm_s8vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_uint16 * scm_u16vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_int16 * scm_s16vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_uint32 * scm_u32vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_int32 * scm_s32vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_uint64 * scm_u64vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
scm_t_int64 * scm_s64vector_writable_elements (SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp)
```
Like `scm_vector_writable_elements` (see Section 6.6.10.4 [Vector Accessing from C], page 189), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

### 7.5.5.3 SRFI-4 - Relation to bytevectors

Guile implements SRFI-4 vectors using bytevectors (see Section 6.6.12 [Bytevectors], page 193). Often when you have a numeric vector, you end up wanting to write its bytes somewhere, or have access to the underlying bytes, or read in bytes from somewhere else. Bytevectors are very good at this sort of thing. But the SRFI-4 APIs are nicer to use when doing number-crunching, because they are addressed by element and not by byte.

So as a compromise, Guile allows all bytevector functions to operate on numeric vectors. They address the underlying bytes in the native endianness, as one would expect.

Following the same reasoning, that it’s just bytes underneath, Guile also allows uniform vectors of a given type to be accessed as if they were of any type. One can fill a `u32vector`, and access its elements with `u8vector-ref`. One can use `f64vector-ref` on bytevectors. It’s all the same to Guile.

In this way, uniform numeric vectors may be written to and read from input/output ports using the procedures that operate on bytevectors.

See Section 6.6.12 [Bytevectors], page 193, for more information.

### 7.5.5.4 SRFI-4 - Guile extensions

Guile defines some useful extensions to SRFI-4, which are not available in the default Guile environment. They may be imported by loading the extensions module:

```scheme
(use-modules (srfi srfi-4 gnu))
```

- `any->u8vector obj` [Scheme Procedure]
- `any->s8vector obj` [Scheme Procedure]
- `any->u16vector obj` [Scheme Procedure]
- `any->s16vector obj` [Scheme Procedure]
- `any->u32vector obj` [Scheme Procedure]
- `any->s32vector obj` [Scheme Procedure]
- `any->u64vector obj` [Scheme Procedure]
- `any->s64vector obj` [Scheme Procedure]
- `any->f32vector obj` [Scheme Procedure]
- `any->f64vector obj` [Scheme Procedure]
- `any->c32vector obj` [Scheme Procedure]
- `any->c64vector obj` [Scheme Procedure]
- `scm_any_to_u8vector (obj)` [C Function]
Chapter 7: Guile Modules

7.5.6 SRFI-6 - Basic String Ports
SRFI-6 defines the procedures open-input-string, open-output-string and get-output-string. These procedures are included in the Guile core, so using this module does not make any difference at the moment. But it is possible that support for SRFI-6 will be factored out of the core library in the future, so using this module does not hurt, after all.

7.5.7 SRFI-8 - receive
receive is a syntax for making the handling of multiple-value procedures easier. It is documented in See Section 6.13.7 [Multiple Values], page 309.

7.5.8 SRFI-9 - define-record-type
This SRFI is a syntax for defining new record types and creating predicate, constructor, and field getter and setter functions. It is documented in the “Data Types” section of the manual (see Section 6.6.16 [SRFI-9 Records], page 218).

7.5.9 SRFI-10 - Hash-Comma Reader Extension
This SRFI implements a reader extension #,() called hash-comma. It allows the reader to give new kinds of objects, for use both in data and as constants or literals in source code. This feature is available with

```
(use-modules (srfi srfi-10))
```

The new read syntax is of the form

```
#,(tag arg...)
```

where tag is a symbol and the args are objects taken as parameters. tags are registered with the following procedure.

**define-reader-ctor** tag proc  [Scheme Procedure]
Register proc as the constructor for a hash-comma read syntax starting with symbol tag, i.e. #,(tag arg...). proc is called with the given arguments (proc arg...) and the object it returns is the result of the read.
For example, a syntax giving a list of \( N \) copies of an object.

\[
\begin{align*}
\text{(define-reader-ctor 'repeat} \\
& \quad \text{(lambda (obj reps)} \\
& \quad \quad \text{(make-list reps obj))})
\end{align*}
\]

\[
\begin{align*}
\text{(display '#,(repeat 99 3))} \\
\quad \downarrow (99 99 99)
\end{align*}
\]

Notice the quote \' when the \#,() is used. The repeat handler returns a list and the program must quote to use it literally, the same as any other list. Ie.

\[
\begin{align*}
\text{(display '#,(repeat 99 3))} \\
\Rightarrow \text{(display '(99 99 99))}
\end{align*}
\]

When a handler returns an object which is self-evaluating, like a number or a string, then there’s no need for quoting, just as there’s no need when giving those directly as literals. For example an addition,

\[
\begin{align*}
\text{(define-reader-ctor 'sum} \\
& \quad \text{(lambda (x y)} \\
& \quad \quad (+ x y))}
\end{align*}
\]

\[
\begin{align*}
\text{(display #,(sum 123 456))} \\
\quad \downarrow 579
\end{align*}
\]

Once \( \text{srfi srfi-10} \) has loaded, \#,() is available globally, there’s no need to use \( \text{srfi srfi-10} \) in later modules. Similarly the tags registered are global and can be used anywhere once registered.

We do not recommend \#,() reader extensions, however, and for three reasons.

First of all, this SRFI is not modular: the tag is matched by name, not as an identifier within a scope. Defining a reader extension in one part of a program can thus affect unrelated parts of a program because the tag is not scoped.

Secondly, reader extensions can be hard to manage from a time perspective: when does the reader extension take effect? See Section 6.10.8 [Eval When], page 279, for more discussion.

Finally, reader extensions can easily produce objects that can’t be reified to an object file by the compiler. For example if you define a reader extension that makes a hash table (see Section 6.6.22 [Hash Tables], page 237), then it will work fine when run with the interpreter, and you think you have a neat hack. But then if you try to compile your program, after wrangling with the eval-when concerns mentioned above, the compiler will carp that it doesn’t know how to serialize a hash table to disk.

In the specific case of hash tables, it would be possible for Guile to know how to pack hash tables into compiled files, but this doesn’t work in general. What if the object you produce is an instance of a record type? Guile would then have to serialize the record type to disk too, and then what happens if the program independently loads the code that defines the record type? Does it define the same type or a different type? Guile’s record types are nominal, not structural, so the answer is not clear at all.

For all of these reasons we recommend macros over reader extensions. Macros fulfill many of the same needs while preserving modular composition, and their interaction with eval-when is well-known. If you need brevity, instead use \text{read-hash-extend} and make
your reader extension expand to a macro invocation. In that way we preserve scoping as much as possible. See Section 6.18.1.6 [Reader Extensions], page 382.

### 7.5.10 SRFI-11 - let-values

This module implements the binding forms for multiple values `let-values` and `let*-values`. These forms are similar to `let` and `let*` (see Section 6.12.2 [Local Bindings], page 295), but they support binding of the values returned by multiple-valued expressions.

Write `(use-modules (srfi srfi-11))` to make the bindings available.

```scheme
(let-values (((x y) (values 1 2))
              ((z f) (values 3 4)))
  (+ x y z f))
⇒
10
```

`let-values` performs all bindings simultaneously, which means that no expression in the binding clauses may refer to variables bound in the same clause list. `let*-values`, on the other hand, performs the bindings sequentially, just like `let*` does for single-valued expressions.

### 7.5.11 SRFI-13 - String Library

The SRFI-13 procedures are always available, See Section 6.6.5 [Strings], page 141.

### 7.5.12 SRFI-14 - Character-set Library

The SRFI-14 data type and procedures are always available, See Section 6.6.4 [Character Sets], page 134.

### 7.5.13 SRFI-16 - case-lambda

SRFI-16 defines a variable-arity `lambda` form, `case-lambda`. This form is available in the default Guile environment. See Section 6.9.5 [Case-lambda], page 255, for more information.

### 7.5.14 SRFI-17 - Generalized set!

This SRFI implements a generalized `set!`, allowing some “referencing” functions to be used as the target location of a `set!`. This feature is available from

```scheme
(use-modules (srfi srfi-17))
```

For example `vector-ref` is extended so that

```scheme
(set! (vector-ref vec idx) new-value)
```

is equivalent to

```scheme
(vector-set! vec idx new-value)
```

The idea is that a `vector-ref` expression identifies a location, which may be either fetched or stored. The same form is used for the location in both cases, encouraging visual clarity. This is similar to the idea of an “lvalue” in C.

The mechanism for this kind of `set!` is in the Guile core (see Section 6.9.8 [Proce-dures with Setters], page 259). This module adds definitions of the following functions as procedures with setters, allowing them to be targets of a `set!`. 

The SRFI specifies setter (see Section 6.9.8 [Procedures with Setters], page 259) as a procedure with setter, allowing the setter for a procedure to be changed, eg. (set! (setter foo) my-new-setter-handler). Currently Guile does not implement this, a setter can only be specified on creation (getter-with-setter below).

### 7.5.15 SRFI-18 - Multithreading support

This is an implementation of the SRFI-18 threading and synchronization library. The functions and variables described here are provided by

```scheme
(use-modules (srfi srfi-18))
```

SRFI-18 defines facilities for threads, mutexes, condition variables, time, and exception handling. Because these facilities are at a higher level than Guile’s primitives, they are implemented as a layer on top of what Guile provides. In particular this means that a Guile mutex is not a SRFI-18 mutex, and a Guile thread is not a SRFI-18 thread, and so on. Guile provides a set of primitives and SRFI-18 is one of the systems built in terms of those primitives.

#### 7.5.15.1 SRFI-18 Threads

Threads created by SRFI-18 differ in two ways from threads created by Guile’s built-in thread functions. First, a thread created by SRFI-18 `make-thread` begins in a blocked state and will not start execution until `thread-start!` is called on it. Second, SRFI-18 threads are constructed with a top-level exception handler that captures any exceptions that are thrown on thread exit.

SRFI-18 threads are disjoint from Guile’s primitive threads. See Section 6.22.1 [Threads], page 436, for more on Guile’s primitive facility.

```scheme
(current-thread)  ; [Function]
```

Returns the thread that called this function. This is the same procedure as the same-named built-in procedure `current-thread` (see Section 6.22.1 [Threads], page 436).

```scheme
(thread? obj)  ; [Function]
```

Returns `#t` if `obj` is a thread, `#f` otherwise. This is the same procedure as the same-named built-in procedure `thread?` (see Section 6.22.1 [Threads], page 436).

```scheme
(make-thread thunk [name])  ; [Function]
```

Call `thunk` in a new thread and with a new dynamic state, returning the new thread and optionally assigning it the object name `name`, which may be any Scheme object. Note that the name `make-thread` conflicts with the `(ice-9 threads)` function `make-thread`. Applications wanting to use both of these functions will need to refer to them by different names.
thread-name thread [Function]
Returns the name assigned to thread at the time of its creation, or #f if it was not given a name.

thread-specific thread [Function]
thread-specific-set! thread obj [Function]
Get or set the “object-specific” property of thread. In Guile’s implementation of SRFI-18, this value is stored as an object property, and will be #f if not set.

thread-start! thread [Function]
Unblocks thread and allows it to begin execution if it has not done so already.

thread-yield! [Function]
If one or more threads are waiting to execute, calling thread-yield! forces an immediate context switch to one of them. Otherwise, thread-yield! has no effect. thread-yield! behaves identically to the Guile built-in function yield.

thread-sleep! timeout [Function]
The current thread waits until the point specified by the time object timeout is reached (see Section 7.5.15.4 [SRFI-18 Time], page 607). This blocks the thread only if timeout represents a point in the future. It is an error for timeout to be #f.

thread-terminate! thread [Function]
Causes an abnormal termination of thread. If thread is not already terminated, all mutexes owned by thread become unlocked/abandoned. If thread is the current thread, thread-terminate! does not return. Otherwise thread-terminate! returns an unspecified value; the termination of thread will occur before thread-terminate! returns. Subsequent attempts to join on thread will cause a “terminated thread exception” to be raised.

thread-terminate! is compatible with the thread cancellation procedures in the core threads API (see Section 6.22.1 [Threads], page 436) in that if a cleanup handler has been installed for the target thread, it will be called before the thread exits and its return value (or exception, if any) will be stored for later retrieval via a call to thread-join!.

thread-join! thread [timeout [timeout-val]] [Function]
Wait for thread to terminate and return its exit value. When a time value timeout is given, it specifies a point in time where the waiting should be aborted. When the waiting is aborted, timeout-val is returned if it is specified; otherwise, a join-timeout-exception exception is raised (see Section 7.5.15.5 [SRFI-18 Exceptions], page 607). Exceptions may also be raised if the thread was terminated by a call to thread-terminate! (terminated-thread-exception will be raised) or if the thread exited by raising an exception that was handled by the top-level exception handler (uncaught-exception will be raised; the original exception can be retrieved using uncaught-exception-reason).

7.5.15.2 SRFI-18 Mutexes

SRFI-18 mutexes are disjoint from Guile’s primitive mutexes. See Section 6.22.5 [Mutexes and Condition Variables], page 442, for more on Guile’s primitive facility.
make-mutex [name]  [Function]
Returns a new mutex, optionally assigning it the object name name, which may be any Scheme object. The returned mutex will be created with the configuration described above.

mutex-name mutex  [Function]
Returns the name assigned to mutex at the time of its creation, or #f if it was not given a name.

mutex-specific mutex  [Function]
Return the “object-specific” property of mutex, or #f if none is set.

mutex-specific-set! mutex obj  [Function]
Set the “object-specific” property of mutex.

mutex-state mutex  [Function]
Returns information about the state of mutex. Possible values are:

- thread t: the mutex is in the locked/owned state and thread t is the owner of the mutex
- symbol not-owned: the mutex is in the locked/not-owned state
- symbol abandoned: the mutex is in the unlocked/abandoned state
- symbol not-abandoned: the mutex is in the unlocked/not-abandoned state

mutex-lock! mutex [timeout [thread]]  [Function]
Lock mutex, optionally specifying a time object timeout after which to abort the lock attempt and a thread thread giving a new owner for mutex different than the current thread.

mutex-unlock! mutex [condition-variable [timeout]]  [Function]
Unlock mutex, optionally specifying a condition variable condition-variable on which to wait, either indefinitely or, optionally, until the time object timeout has passed, to be signalled.

7.5.15.3 SRFI-18 Condition variables

SRFI-18 does not specify a “wait” function for condition variables. Waiting on a condition variable can be simulated using the SRFI-18 mutex-unlock! function described in the previous section.

SRFI-18 condition variables are disjoint from Guile’s primitive condition variables. See Section 6.22.5 [Mutexes and Condition Variables], page 442, for more on Guile’s primitive facility.

condition-variable? obj  [Function]
Returns #t if obj is a condition variable, #f otherwise.

make-condition-variable [name]  [Function]
Returns a new condition variable, optionally assigning it the object name name, which may be any Scheme object.
**condition-variable-name**  
*condition-variable*  
[Function]  
Returns the name assigned to *condition-variable* at the time of its creation, or #f if it was not given a name.

**condition-variable-specific**  
*condition-variable*  
[Function]  
Return the “object-specific” property of *condition-variable*, or #f if none is set.

**condition-variable-specific-set!**  
*condition-variable*  
*obj*  
[Function]  
Set the “object-specific” property of *condition-variable*.

**condition-variable-signal!**  
*condition-variable*  
[Function]  
**condition-variable-broadcast!**  
*condition-variable*  
[Function]  
Wake up one thread that is waiting for *condition-variable*, in the case of  
*condition-variable-signal!*, or all threads waiting for it, in the case of  
*condition-variable-broadcast!*.

### 7.5.15.4 SRFI-18 Time

The SRFI-18 time functions manipulate time in two formats: a “time object” type that represents an absolute point in time in some implementation-specific way; and the number of seconds since some unspecified “epoch”. In Guile’s implementation, the epoch is the Unix epoch, 00:00:00 UTC, January 1, 1970.

**current-time**  
[Function]  
Return the current time as a time object. This procedure replaces the procedure of the same name in the core library, which returns the current time in seconds since the epoch.

**time?**  
*obj*  
[Function]  
Returns #t if *obj* is a time object, #f otherwise.

**time->seconds**  
*time*  
[Function]  
Convert between time objects and numerical values representing the number of seconds since the epoch. When converting from a time object to seconds, the return value is the number of seconds between *time* and the epoch. When converting from seconds to a time object, the return value is a time object that represents a time *seconds* seconds after the epoch.

### 7.5.15.5 SRFI-18 Exceptions

SRFI-18 exceptions are identical to the exceptions provided by Guile’s implementation of SRFI-34. The behavior of exception handlers invoked to handle exceptions thrown from SRFI-18 functions, however, differs from the conventional behavior of SRFI-34 in that the continuation of the handler is the same as that of the call to the function. Handlers are called in a tail-recursive manner; the exceptions do not “bubble up”.

**current-exception-handler**  
[Function]  
Returns the current exception handler.
with-exception-handler  handler thunk  [Function]
Installs handler as the current exception handler and calls the procedure thunk with
no arguments, returning its value as the value of the exception. handler must be a
procedure that accepts a single argument. The current exception handler at the time
this procedure is called will be restored after the call returns.

raise  obj  [Function]
Raise obj as an exception. This is the same procedure as the same-named procedure
defined in SRFI 34.

join-timeout-exception?  obj  [Function]
Returns #t if obj is an exception raised as the result of performing a timed join on a
thread that does not exit within the specified timeout, #f otherwise.

abandoned-mutex-exception?  obj  [Function]
Returns #t if obj is an exception raised as the result of attempting to lock a mutex
that has been abandoned by its owner thread, #f otherwise.

terminated-thread-exception?  obj  [Function]
Returns #t if obj is an exception raised as the result of joining on a thread that exited
as the result of a call to thread-terminate!.

uncaught-exception?  obj
uncaught-exception-reason  exc
uncaught-exception? returns #t if obj is an exception thrown as the result of joining
a thread that exited by raising an exception that was handled by the top-level ex-
ception handler installed by make-thread. When this occurs, the original exception
is preserved as part of the exception thrown by thread-join! and can be accessed
by calling uncaught-exception-reason on that exception. Note that because this
exception-preservation mechanism is a side-effect of make-thread, joining on threads
that exited as described above but were created by other means will not raise this
uncaught-exception error.

7.5.16 SRFI-19 - Time/Date Library
This is an implementation of the SRFI-19 time/date library. The functions and variables
described here are provided by

(use-modules (srfi srfi-19))

7.5.16.1 SRFI-19 Introduction
This module implements time and date representations and calculations, in various time
systems, including Coordinated Universal Time (UTC) and International Atomic Time
(TAI).

For those not familiar with these time systems, TAI is based on a fixed length second
derived from oscillations of certain atoms. UTC differs from TAI by an integral number
of seconds, which is increased or decreased at announced times to keep UTC aligned to a
mean solar day (the orbit and rotation of the earth are not quite constant).

So far, only increases in the TAI ↔ UTC difference have been needed. Such an increase
is a “leap second”, an extra second of TAI introduced at the end of a UTC day. When
working entirely within UTC this is never seen, every day simply has 86400 seconds. But when converting from TAI to a UTC date, an extra 23:59:60 is present, where normally a day would end at 23:59:59. Effectively the UTC second from 23:59:59 to 00:00:00 has taken two TAI seconds.

In the current implementation, the system clock is assumed to be UTC, and a table of leap seconds in the code converts to TAI. See comments in `srfi-19.scm` for how to update this table.

Also, for those not familiar with the terminology, a *Julian Day* represents a point in time as a real number of days since -4713-11-24T12:00:00Z, i.e. midday UT on 24 November 4714 BC in the proleptic Gregorian calendar (1 January 4713 BC in the proleptic Julian calendar).

A *Modified Julian Day* represents a point in time as a real number of days since 1858-11-17T00:00:00Z, i.e. midnight UT on Wednesday 17 November AD 1858. That time is julian day 2400000.5.

### 7.5.16.2 SRFI-19 Time

A *time* object has type, seconds and nanoseconds fields representing a point in time starting from some epoch. This is an arbitrary point in time, not just a time of day. Although times are represented in nanoseconds, the actual resolution may be lower.

The following variables hold the possible time types. For instance `(current-time time-process)` would give the current CPU process time.

```scheme
(time-utc)  [Variable]
  Universal Coordinated Time (UTC).
(time-tai)  [Variable]
  International Atomic Time (TAI).
(time-monotonic)  [Variable]
  Monotonic time, meaning a monotonically increasing time starting from an unspecified epoch.
  Note that in the current implementation time-monotonic is the same as time-tai, and unfortunately is therefore affected by adjustments to the system clock. Perhaps this will change in the future.
(time-duration)  [Variable]
  A duration, meaning simply a difference between two times.
(time-process)  [Variable]
  CPU time spent in the current process, starting from when the process began.
(time-thread)  [Variable]
  CPU time spent in the current thread. Not currently implemented.
```

```scheme
(time? obj)  [Function]
  Return #t if obj is a time object, or #f if not.

(make-time type nanoseconds seconds)  [Function]
  Create a time object with the given type, seconds and nanoseconds.
```
Get or set the type, seconds or nanoseconds fields of a time object. set-time-type! merely changes the field, it doesn’t convert the time value. For conversions, see Section 7.5.16.4 [SRFI-19 Time/Date conversions], page 612.

copy-time time
Return a new time object, which is a copy of the given time.

current-time [type]
Return the current time of the given type. The default type is time-utc.
Note that the name current-time conflicts with the Guile core current-time function (see Section 7.2.5 [Time], page 509) as well as the SRFI-18 current-time function (see Section 7.5.15.4 [SRFI-18 Time], page 607). Applications wanting to use more than one of these functions will need to refer to them by different names.

time-resolution [type]
Return the resolution, in nanoseconds, of the given time type. The default type is time-utc.

time<=? t1 t2
time>? t1 t2
time>=? t1 t2
time>? t1 t2
Return #t or #f according to the respective relation between time objects t1 and t2. t1 and t2 must be the same time type.

time-difference t1 t2
time-difference! t1 t2
Return a time object of type time-duration representing the period between t1 and t2. t1 and t2 must be the same time type.

time-difference returns a new time object, time-difference! may modify t1 to form its return.

add-duration time duration
add-duration! time duration
subtract-duration time duration
subtract-duration! time duration
Return a time object which is time with the given duration added or subtracted. duration must be a time object of type time-duration.

add-duration and subtract-duration return a new time object. add-duration! and subtract-duration! may modify the given time to form their return.
7.5.16.3 SRFI-19 Date

A date object represents a date in the Gregorian calendar and a time of day on that date in some timezone.

The fields are year, month, day, hour, minute, second, nanoseconds and timezone. A date object is immutable, its fields can be read but they cannot be modified once the object is created.

Historically, the Gregorian calendar was only used from the latter part of the year 1582 onwards, and not until even later in many countries. Prior to that most countries used the Julian calendar. SRFI-19 does not deal with the Julian calendar at all, and so does not reflect this historical calendar reform. Instead it projects the Gregorian calendar back proleptically as far as necessary. When dealing with historical data, especially prior to the British Empire’s adoption of the Gregorian calendar in 1752, one should be mindful of which calendar is used in each context, and apply non-SRFI-19 facilities to convert where necessary.

**date?** obj

Return #t if obj is a date object, or #f if not.

**make-date** nsecs seconds minutes hours date month year zone-offset

Create a new date object.

**date-nanosecond** date

Nanoseconds, 0 to 999999999.

**date-second** date

Seconds, 0 to 59, or 60 for a leap second. 60 is never seen when working entirely within UTC, it’s only when converting to or from TAI.

**date-minute** date

Minutes, 0 to 59.

**date-hour** date

Hour, 0 to 23.

**date-day** date

Day of the month, 1 to 31 (or less, according to the month).

**date-month** date

Month, 1 to 12.

**date-year** date

Year, eg. 2003. Dates B.C. are negative, eg. −46 is 46 B.C. There is no year 0, year −1 is followed by year 1.

**date-zone-offset** date

Time zone, an integer number of seconds east of Greenwich.

**date-year-day** date

Day of the year, starting from 1 for 1st January.
date-week-day date
   Day of the week, starting from 0 for Sunday.

date-week-number date dstartw
   Week of the year, ignoring a first partial week. dstartw is the day of the week which is taken to start a week, 0 for Sunday, 1 for Monday, etc.

current-date [tz-offset]
   Return a date object representing the current date/time, in UTC offset by tz-offset. tz-offset is seconds east of Greenwich and defaults to the local timezone.

current-julian-day
   Return the current Julian Day.

current-modified-julian-day
   Return the current Modified Julian Day.

7.5.16.4 SRFI-19 Time/Date conversions

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]

julian-day->date jdn [tz-offset]
   [Function]
julian-day->time-monotonic jdn
   [Function]
julian-day->time-tai jdn
   [Function]
julian-day->time-utc jdn
   [Function]

modified-julian-day->date jdn [tz-offset]
   [Function]
modified-julian-day->time-monotonic jdn
   [Function]
modified-julian-day->time-tai jdn
   [Function]
modified-julian-day->time-utc jdn
   [Function]

time-monotonic->date time [tz-offset]
   [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]

time-tai->date time [tz-offset]
   [Function]
time-tai->julian-day time
   [Function]
time-tai->modified-julian-day time
   [Function]
time-tai->time-monotonic time
   [Function]
time-tai->time-monotonic! time
   [Function]
time-tai->time-utc time
   [Function]
time-tai->time-utc! time
   [Function]

time-utc->date time [tz-offset]
   [Function]
time-utc->julian-day time
   [Function]
Convert between dates, times and days of the respective types. For instance `time-tai->time-utc` accepts a `time` object of type `time-tai` and returns an object of type `time-utc`.

The `!` variants may modify their `time` argument to form their return. The plain functions create a new object.

For conversions to dates, `tz-offset` is seconds east of Greenwich. The default is the local timezone, at the given time, as provided by the system, using `localtime` (see Section 7.2.5 [Time], page 509).

On 32-bit systems, `localtime` is limited to a 32-bit `time_t`, so a default `tz-offset` is only available for times between Dec 1901 and Jan 2038. For prior dates an application might like to use the value in 1902, though some locations have zone changes prior to that. For future dates an application might like to assume today’s rules extend indefinitely. But for correct daylight savings transitions it will be necessary to take an offset for the same day and time but a year in range and which has the same starting weekday and same leap/non-leap (to support rules like last Sunday in October).

### 7.5.16.5 SRFI-19 Date to string

`date->string date [format]` [Function]

Convert a date to a string under the control of a format. `format` should be a string containing ‘~’ escapes, which will be expanded as per the following conversion table. The default `format` is ‘~c’, a locale-dependent date and time.

Many of these conversion characters are the same as POSIX `strftime` (see Section 7.2.5 [Time], page 509), but there are some extras and some variations.

- `~` literal ‘
- `a` locale abbreviated weekday, eg. ‘Sun’
- `A` locale full weekday, eg. ‘Sunday’
- `b` locale abbreviated month, eg. ‘Jan’
- `B` locale full month, eg. ‘January’
- `c` locale date and time, eg. ‘Fri Jul 14 20:28:42-0400 2000’
- `d` day of month, zero padded, ‘01’ to ‘31’
- `e` day of month, blank padded, ‘ 1’ to ‘31’
- `f` seconds and fractional seconds, with locale decimal point, eg. ‘5.2’
- `h` same as ‘b’
- `H` hour, 24-hour clock, zero padded, ‘00’ to ‘23’
- `I` hour, 12-hour clock, zero padded, ‘01’ to ‘12’
- `j` day of year, zero padded, ‘001’ to ‘366’
- `k` hour, 24-hour clock, blank padded, ‘ 0’ to ‘23’
~l hour, 12-hour clock, blank padded, ‘1’ to ‘12’
~m month, zero padded, ‘01’ to ‘12’
~M minute, zero padded, ‘00’ to ‘59’
~n newline
~N nanosecond, zero padded, ‘000000000’ to ‘999999999’
~p locale AM or PM
~r time, 12 hour clock, ‘I:~M:~S ~p’
~s number of full seconds since “the epoch” in UTC
~S second, zero padded ‘00’ to ‘60’
(usual limit is 59, 60 is a leap second)
~t horizontal tab character
~T time, 24 hour clock, ‘~H:~M:~S’
~U week of year, Sunday first day of week, ‘00’ to ‘52’
~V week of year, Monday first day of week, ‘01’ to ‘53’
~w day of week, 0 for Sunday, ‘0’ to ‘6’
~W week of year, Monday first day of week, ‘00’ to ‘52’
~y year, two digits, ‘00’ to ‘99’
~Y year, full, eg. ‘2003’
~z time zone, RFC-822 style
~Z time zone symbol (not currently implemented)
~1 ISO-8601 date, ‘~Y-~m-~d’
~2 ISO-8601 time+zone, ‘~H:~M:~S~z’
~3 ISO-8601 time, ‘~H:~M:~S’
~4 ISO-8601 date/time+zone, ‘~Y-~m-~dT~H:~M:~S~z’
~5 ISO-8601 date/time, ‘~Y-~m-~dT~H:~M:~S’

Conversions ‘~D’, ‘~x’ and ‘~X’ are not currently described here, since the specification and reference implementation differ.

Conversion is locale-dependent on systems that support it (see Section 6.25.5 [Accessing Locale Information], page 462). See Section 7.2.13 [Locales], page 542, for information on how to change the current locale.

7.5.16.6 SRFI-19 String to date

string->date input template  [Function]
Convert an input string to a date under the control of a template string. Return a newly created date object.

Literal characters in template must match characters in input and ‘~’ escapes must match the input forms described in the table below. “Skip to” means characters up to one of the given type are ignored, or “no skip” for no skipping. “Read” is what’s then read, and “Set” is the field affected in the date object.

For example ‘~Y’ skips input characters until a digit is reached, at which point it expects a year and stores that to the year field of the date.

<table>
<thead>
<tr>
<th>Skip to</th>
<th>Read</th>
<th>Set</th>
</tr>
</thead>
</table>
Chapter 7: Guile Modules

---

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>no skip</td>
<td>nothing</td>
</tr>
<tr>
<td>~a</td>
<td>char-alphabetic?</td>
<td>locale abbreviated weekday name</td>
</tr>
<tr>
<td>~A</td>
<td>char-alphabetic?</td>
<td>locale full weekday name</td>
</tr>
<tr>
<td>~b</td>
<td>char-alphabetic?</td>
<td>locale abbreviated month name</td>
</tr>
<tr>
<td>~B</td>
<td>char-alphabetic?</td>
<td>locale full month name</td>
</tr>
<tr>
<td>~d</td>
<td>char-numeric?</td>
<td>day of month</td>
</tr>
<tr>
<td>~e</td>
<td>no skip</td>
<td>day of month, blank padded</td>
</tr>
<tr>
<td>~h</td>
<td>same as ‘~b’</td>
<td>date-day</td>
</tr>
<tr>
<td>~H</td>
<td>char-numeric?</td>
<td>hour</td>
</tr>
<tr>
<td>~k</td>
<td>no skip</td>
<td>hour, blank padded</td>
</tr>
<tr>
<td>~m</td>
<td>char-numeric?</td>
<td>month</td>
</tr>
<tr>
<td>~M</td>
<td>char-numeric?</td>
<td>minute</td>
</tr>
<tr>
<td>~N</td>
<td>char-numeric?</td>
<td>nanosecond</td>
</tr>
<tr>
<td>~S</td>
<td>char-numeric?</td>
<td>second</td>
</tr>
<tr>
<td>~y</td>
<td>no skip</td>
<td>2-digit year</td>
</tr>
<tr>
<td>~Y</td>
<td>char-numeric?</td>
<td>year</td>
</tr>
<tr>
<td>~z</td>
<td>no skip</td>
<td>time zone</td>
</tr>
</tbody>
</table>

Notice that the weekday matching forms don’t affect the date object returned, instead the weekday will be derived from the day, month and year.

Conversion is locale-dependent on systems that support it (see Section 6.25.5 [Accessing Locale Information], page 462). See Section 7.2.13 [Locales], page 542, for information on how to change the current locale.

7.5.17 SRFI-23 - Error Reporting
The SRFI-23 error procedure is always available.

7.5.18 SRFI-26 - specializing parameters
This SRFI provides a syntax for conveniently specializing selected parameters of a function. It can be used with,
(use-modules (srfi srfi-26))

cut slot1 slot2 ... [library syntax]
cute slot1 slot2 ... [library syntax]

Return a new procedure which will make a call (slot1 slot2 ...) but with selected parameters specialized to given expressions.

An example will illustrate the idea. The following is a specialization of write, sending output to my-output-port,

(cut write <> my-output-port)
⇒
(lambda (obj) (write obj my-output-port))

The special symbol <> indicates a slot to be filled by an argument to the new procedure. my-output-port on the other hand is an expression to be evaluated and passed, ie. it specializes the behaviour of write.

<> A slot to be filled by an argument from the created procedure. Arguments are assigned to <> slots in the order they appear in the cut form, there’s no way to re-arrange arguments.

The first argument to cut is usually a procedure (or expression giving a procedure), but <> is allowed there too. For example,

(cut <> 1 2 3)
⇒
(lambda (proc) (proc 1 2 3))

<...> A slot to be filled by all remaining arguments from the new procedure. This can only occur at the end of a cut form.

For example, a procedure taking a variable number of arguments like max but in addition enforcing a lower bound,

(define my-lower-bound 123)

(cut max my-lower-bound <...>)
⇒
(lambda arglist (apply max my-lower-bound arglist))

For cut the specializing expressions are evaluated each time the new procedure is called. For cute they’re evaluated just once, when the new procedure is created. The name cute stands for “cut with evaluated arguments”. In all cases the evaluations take place in an unspecified order.

The following illustrates the difference between cut and cute,

(cut format <> "the time is \~s" (current-time))
⇒
(lambda (port) (format port "the time is \~s" (current-time)))

(cute format <> "the time is \~s" (current-time))
⇒
(let ((val (current-time))

(lambda (port) (format port "the time is \~s" val))
(There’s no provision for a mixture of cut and cute where some expressions would be evaluated every time but others evaluated only once.)
cut is really just a shorthand for the sort of lambda forms shown in the above examples. But notice cut avoids the need to name unspecialized parameters, and is more compact. Use in functional programming style or just with map, for-each or similar is typical.

(map (cut * 2 <> ) '(1 2 3 4))

(for-each (cut write <> my-port) my-list)

7.5.19 SRFI-27 - Sources of Random Bits
This subsection is based on the specification of SRFI-27 (http://srfi.schemers.org/srfi-27/srfi-27.html) written by Sebastian Egner.

This SRFI provides access to a (pseudo) random number generator; for Guile’s built-in random number facilities, which SRFI-27 is implemented upon, See Section 6.6.2.14 [Random], page 127. With SRFI-27, random numbers are obtained from a random source, which encapsulates a random number generation algorithm and its state.

7.5.19.1 The Default Random Source

random-integer n  [Function]
Return a random number between zero (inclusive) and n (exclusive), using the default random source. The numbers returned have a uniform distribution.

random-real       [Function]
Return a random number in (0,1), using the default random source. The numbers returned have a uniform distribution.

default-random-source [Function]
A random source from which random-integer and random-real have been derived using random-source-make-integers and random-source-make-reals (see Section 7.5.19.3 [SRFI-27 Random Number Generators], page 618, for those procedures). Note that an assignment to default-random-source does not change random-integer or random-real; it is also strongly recommended not to assign a new value.

7.5.19.2 Random Sources

make-random-source [Function]
Create a new random source. The stream of random numbers obtained from each random source created by this procedure will be identical, unless its state is changed by one of the procedures below.

random-source? object [Function]
Tests whether object is a random source. Random sources are a disjoint type.

random-source-randomize! source [Function]
Attempt to set the state of the random source to a truly random value. The current implementation uses a seed based on the current system time.
**random-source-pseudo-randomize!**  
source i j  

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 provides a mechanism to obtain a large number of independent random sources (usually all derived from the same backbone generator), indexed by two integers. In contrast to **random-source-randomize!**, this procedure is entirely deterministic.

The state associated with a random state can be obtained and reinstated with the following procedures:

**random-source-state-ref**  
source  

**random-source-state-set!**  
source state  

Get and set the state of a random source. No assumptions should be made about the nature of the state object, besides it having an external representation (i.e. it can be passed to write and subsequently read back).

### 7.5.19.3 Obtaining random number generator procedures

**random-source-make-integers**  
source  

Obtains a procedure to generate random integers using the random source source. The returned procedure takes a single argument n, which must be a positive integer, and returns the next uniformly distributed random integer from the interval \{0, ..., n-1\} by advancing the state of source.

If an application obtains and uses several generators for the same random source source, a call to any of these generators advances the state of source. Hence, the generators do not produce the same sequence of random integers each but rather share a state. This also holds for all other types of generators derived from a fixed random sources.

While the SRFI text specifies that “Implementations that support concurrency make sure that the state of a generator is properly advanced”, this is currently not the case in Guile’s implementation of SRFI-27, as it would cause a severe performance penalty. So in multi-threaded programs, you either must perform locking on random sources shared between threads yourself, or use different random sources for multiple threads.

**random-source-make-reals**  
source  

**random-source-make-reals**  
source unit  

Obtains a procedure to generate random real numbers \(0 < x < 1\) using the random source source. The procedure rand is called without arguments.

The optional parameter unit determines the type of numbers being produced by the returned procedure and the quantization of the output. unit must be a number such that \(0 < unit < 1\). The numbers created by the returned procedure are of the same numerical type as unit and the potential output values are spaced by at most unit. One can imagine rand to create numbers as \(x \times unit\) where \(x\) is a random integer in \(\{1, ..., \text{floor}(1/unit)-1\}\). Note, however, that this need not be the way the values are actually created and that the actual resolution of rand can be much higher than unit.

In case unit is absent it defaults to a reasonably small value (related to the width of the mantissa of an efficient number format).
7.5.20 SRFI-28 - Basic Format Strings

SRFI-28 provides a basic format procedure that provides only the ~a, ~s, ~%, and ~~ format specifiers. You can import this procedure by using:

(use-modules (srfi srfi-28))

format message arg ... [Scheme Procedure]

Returns a formatted message, using message as the format string, which can contain the following format specifiers:

~a Insert the textual representation of the next arg, as if printed by display.
~s Insert the textual representation of the next arg, as if printed by write.
~% Insert a newline.
~~ Insert a tilde.

This procedure is the same as calling simple-format (see Section 6.14.5 [Simple Output], page 335) with #f as the destination.

7.5.21 SRFI-30 - Nested Multi-line Comments

Starting from version 2.0, Guile’s read supports SRFI-30/R6RS nested multi-line comments by default, Section 6.18.1.3 [Block Comments], page 381.

7.5.22 SRFI-31 - A special form ‘rec’ for recursive evaluation

SRFI-31 defines a special form that can be used to create self-referential expressions more conveniently. The syntax is as follows:

<rec expression> --> (rec <variable> <expression>)
<rec expression> --> (rec (<variable>+) <body>)

The first syntax can be used to create self-referential expressions, for example:

guile> (define tmp (rec ones (cons 1 (delay ones))))

guile> (tmp 42 3)
424242
guile>

7.5.23 SRFI-34 - Exception handling for programs

Guile provides an implementation of SRFI-34’s exception handling mechanisms (http://srfi.schemers.org/srfi-34/srfi-34.html) as an alternative to its own built-in mechanisms (see Section 6.13.8 [Exceptions], page 311). It can be made available as follows:

(use-modules (srfi srfi-34))
7.5.24 SRFI-35 - Conditions

SRFI-35 (http://srfi.schemers.org/srfi-35/srfi-35.html) implements conditions, a data structure akin to records designed to convey information about exceptional conditions between parts of a program. It is normally used in conjunction with SRFI-34’s `raise`:

```scheme
(rraise (condition (&message (message "An error occurred")))
```

Users can define condition types containing arbitrary information. Condition types may inherit from one another. This allows the part of the program that handles (or “catches”) conditions to get accurate information about the exceptional condition that arose.

SRFI-35 conditions are made available using:

```scheme
(use-modules (srfi srfi-35))
```

The procedures available to manipulate condition types are the following:

- `make-condition-type id parent field-names` [Scheme Procedure]
  Return a new condition type named `id`, inheriting from `parent`, and with the fields whose names are listed in `field-names`. `field-names` must be a list of symbols and must not contain names already used by `parent` or one of its supertypes.

- `condition-type? obj` [Scheme Procedure]
  Return true if `obj` is a condition type.

Conditions can be created and accessed with the following procedures:

- `make-condition type . field+value` [Scheme Procedure]
  Return a new condition of type `type` with fields initialized as specified by `field+value`, a sequence of field names (symbols) and values as in the following example:

  ```scheme
  (let ((&ct (make-condition-type 'foo &condition '(a b c))))
    (make-condition &ct 'a 1 'b 2 'c 3))
  ```

  Note that all fields of `type` and its supertypes must be specified.

- `make-compound-condition condition1 condition2 . . .` [Scheme Procedure]
  Return a new compound condition composed of `condition1 condition2 . . .`. The returned condition has the type of each condition of `condition1 condition2 . . .` (per `condition-has-type?`).

- `condition-has-type? c type` [Scheme Procedure]
  Return true if condition `c` has type `type`.

- `condition-ref c field-name` [Scheme Procedure]
  Return the value of the field named `field-name` from condition `c`.
  If `c` is a compound condition and several underlying condition types contain a field named `field-name`, then the value of the first such field is returned, using the order in which conditions were passed to `make-compound-condition`.

- `extract-condition c type` [Scheme Procedure]
  Return a condition of condition type `type` with the field values specified by `c`.
  If `c` is a compound condition, extract the field values from the subcondition belonging to `type` that appeared first in the call to `make-compound-condition` that created the condition.
Convenience macros are also available to create condition types and conditions.

**define-condition-type** type supertype predicate field-spec...  
(library syntax)
Define a new condition type named type that inherits from supertype. In addition, bind predicate to a type predicate that returns true when passed a condition of type type or any of its subtypes. field-spec must have the form (field accessor) where field is the name of field of type and accessor is the name of a procedure to access field field in conditions of type type.

The example below defines condition type &foo, inheriting from &condition with fields a, b and c:

```
(define-condition-type &foo &condition
  foo-condition?
  (a foo-a)
  (b foo-b)
  (c foo-c))
```

**condition** type-field-binding1 type-field-binding2 ...  
(library syntax)
Return a new condition or compound condition, initialized according to type-field-binding1 type-field-binding2 ... . Each type-field-binding must have the form (type field-specs...), where type is the name of a variable bound to a condition type; each field-spec must have the form (field-name value) where field-name is a symbol denoting the field being initialized to value. As for make-condition, all fields must be specified.

The following example returns a simple condition:

```
(condition (&message (message "An error occurred")))
```

The one below returns a compound condition:

```
(condition (&message (message "An error occurred"))
  (&serious))
```

Finally, SRFI-35 defines a several standard condition types.

**&condition**  
[Variable]
This condition type is the root of all condition types. It has no fields.

**&message**  
[Variable]
A condition type that carries a message describing the nature of the condition to humans.

**message-condition?** c  
[Scheme Procedure]
Return true if c is of type &message or one of its subtypes.

**condition-message** c  
[Scheme Procedure]
Return the message associated with message condition c.

**&serious**  
[Variable]
This type describes conditions serious enough that they cannot safely be ignored. It has no fields.
serious-condition? c
    [Scheme Procedure]
    Return true if c is of type &serious or one of its subtypes.

&error
    [Variable]
    This condition describes errors, typically caused by something that has gone wrong
    in the interaction of the program with the external world or the user.

error? c
    [Scheme Procedure]
    Return true if c is of type &error or one of its subtypes.

7.5.25 SRFI-37 - args-fold

This is a processor for GNU getopt_long-style program arguments. It provides an alternative,
less declarative interface than getopt-long in (ice-9 getopt-long) (see Section 7.4
[The (ice-9 getopt-long) Module], page 571). Unlike getopt-long, it supports repeated
options and any number of short and long names per option. Access it with:

(use-modules (srfi srfi-37))

SRFI-37 principally provides an option type and the args-fold function. To use the
library, create a set of options with option and use it as a specification for invoking
args-fold.

Here is an example of a simple argument processor for the typical '--version' and
'--help' options, which returns a backwards list of files given on the command line:

(args-fold (cdr (program-arguments))
  (let ((display-and-exit-proc
    (lambda (msg)
      (lambda (opt name arg loads)
        (display-and-exit-proc (msg) (quit))))))
    (list (option '(#v "version") #f #f
      (display-and-exit-proc "Foo version 42.0
    "
      (option '(#\h "help") #f #f
        (display-and-exit-proc
          "Usage: foo scheme-file ...
          ")))
    (lambda (opt name arg loads)
      (error "Unrecognized option " name)))
    (lambda (op loads) (cons op loads))
    ()
  ))

option names required-arg? optional-arg? processor
    [Scheme Procedure]
    Return an object that specifies a single kind of program option.

    names is a list of command-line option names, and should consist of characters for
traditional getopt short options and strings for getopt_long-style long options.

    required-arg? and optional-arg? are mutually exclusive; one or both must be #f. If
    required-arg?, the option must be followed by an argument on the command line,
such as '--opt=value' for long options, or an error will be signalled. If optional-arg?,
an argument will be taken if available.

    processor is a procedure that takes at least 3 arguments, called when args-fold
    encounters the option: the containing option object, the name used on the command
line, and the argument given for the option (or \#f if none). The rest of the arguments are \texttt{args-fold} “seeds”, and the processor should return seeds as well.

\begin{verbatim}
option-names opt        [Scheme Procedure]
option-required-arg? opt [Scheme Procedure]
option-optional-arg? opt [Scheme Procedure]
option-processor opt    [Scheme Procedure]
\end{verbatim}

Return the specified field of \texttt{opt}, an option object, as described above for \texttt{option}.

\begin{verbatim}
args-fold args options unrecognized-option-proc
        [Scheme Procedure]
        
operand-proc seed . . .
\end{verbatim}

Process \texttt{args}, a list of program arguments such as that returned by \texttt{(cdr (program-arguments))}, in order against \texttt{options}, a list of option objects as described above. All functions called take the “seeds”, or the last multiple-values as multiple arguments, starting with \texttt{seed . . .}, and must return the new seeds. Return the final seeds.

Call \texttt{unrecognized-option-proc}, which is like an option object’s processor, for any options not found in \texttt{options}.

Call \texttt{operand-proc} with any items on the command line that are not named options. This includes arguments after ‘--’. It is called with the argument in question, as well as the seeds.

\subsection*{7.5.26 SRFI-38 - External Representation for Data With Shared Structure}

This subsection is based on the specification of SRFI-38 (http://srfi.schemers.org/srfi-38/srfi-38.html) written by Ray Dillinger.

This SRFI creates an alternative external representation for data written and read using \texttt{write-with-shared-structure} and \texttt{read-with-shared-structure}. It is identical to the grammar for external representation for data written and read with \texttt{write} and \texttt{read} given in section 7 of R5RS, except that the single production

\begin{verbatim}<datum> --> <simple datum> | <compound datum>
\end{verbatim}

is replaced by the following five productions:

\begin{verbatim}<datum> --> <defining datum> | <nondefining datum> | <defined datum>
<defining datum> --> #<indexnum>=<nondefining datum>
<defined datum> --> #<indexnum>#
<nondefining datum> --> <simple datum> | <compound datum>
<indexnum> --> <digit 10>+
\end{verbatim}

\begin{verbatim}
write-with-shared-structure obj        [Scheme procedure]
write-with-shared-structure obj port   [Scheme procedure]
write-with-shared-structure obj port optarg [Scheme procedure]
\end{verbatim}

Writes an external representation of \texttt{obj} to the given port. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the \#\ notation.
Objects which denote locations rather than values (cons cells, vectors, and non-zero-length strings in R5RS scheme; also Guile’s structs, bytevectors and ports and hash-tables), if they appear at more than one point in the data being written, are preceded by ‘#N=’ the first time they are written and replaced by ‘#N#’ all subsequent times they are written, where N is a natural number used to identify that particular object. If objects which denote locations occur only once in the structure, then \texttt{write-with-shared-structure} must produce the same external representation for those objects as \texttt{write}.

\texttt{write-with-shared-structure} terminates in finite time and produces a finite representation when writing finite data.

\texttt{write-with-shared-structure} returns an unspecified value. The \texttt{port} argument may be omitted, in which case it defaults to the value returned by \texttt{(current-output-port)}. The \texttt{optarg} argument may also be omitted. If present, its effects on the output and return value are unspecified but \texttt{write-with-shared-structure} must still write a representation that can be read by \texttt{read-with-shared-structure}. Some implementations may wish to use \texttt{optarg} to specify formatting conventions, numeric radices, or return values. Guile’s implementation ignores \texttt{optarg}.

For example, the code

\begin{verbatim}
  (begin (define a (cons 'val1 'val2))
         (set-cdr! a a)
         (write-with-shared-structure a))
\end{verbatim}

should produce the output \#1=(val1 . #1#). This shows a cons cell whose \texttt{cdr} contains itself.

\texttt{read-with-shared-structure} \hspace{1cm} [Scheme procedure]
\texttt{read-with-shared-structure} \hspace{1cm} [Scheme procedure]
\texttt{read-with-shared-structure} \texttt{port}
\texttt{read-with-shared-structure} converts the external representations of Scheme objects produced by \texttt{write-with-shared-structure} into Scheme objects. That is, it is a parser for the nonterminal \texttt{<datum>} in the augmented external representation grammar defined above. \texttt{read-with-shared-structure} returns the next object parsable from the given input port, updating \texttt{port} to point to the first character past the end of the external representation of the object.

If an end-of-file is encountered in the input before any characters are found that can begin an object, then an end-of-file object is returned. The port remains open, and further attempts to read it (by \texttt{read-with-shared-structure} or \texttt{read} will also return an end-of-file object. If an end of file is encountered after the beginning of an object’s external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The \texttt{port} argument may be omitted, in which case it defaults to the value returned by \texttt{(current-input-port)}. It is an error to read from a closed port.

\subsection{7.5.27 SRFI-39 - Parameters}

This SRFI adds support for dynamically-scoped parameters. SRFI 39 is implemented in the Guile core; there’s no module needed to get SRFI-39 itself. Parameters are documented in Section 6.13.12 [Parameters], page 324.
This module does export one extra function: `with-parameters*`. This is a Guile-specific addition to the SRFI, similar to the core `with-fluids*` (see Section 6.13.11 [Fluids and Dynamic States], page 321).

```scheme
with-parameters* param-list value-list thunk  [Function]
   Establish a new dynamic scope, as per parameterize above, taking parameters from param-list and corresponding values from value-list. A call (thunk) is made in the new scope and the result from that thunk is the return from with-parameters*.
```

### 7.5.28 SRFI-41 - Streams


This SRFI implements streams, sometimes called lazy lists, a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again. SRFI-41 can be made available with:

```scheme
(use-modules (srfi srfi-41))
```

#### 7.5.28.1 SRFI-41 Stream Fundamentals

SRFI-41 Streams are based on two mutually-recursive abstract data types: An object of the `stream` abstract data type is a promise that, when forced, is either `stream-null` or is an object of type `stream-pair`. An object of the `stream-pair` abstract data type contains a `stream-car` and a `stream-cdr`, which must be a `stream`. The essential feature of streams is the systematic suspensions of the recursive promises between the two data types.

The object stored in the `stream-car` of a `stream-pair` is a promise that is forced the first time the `stream-car` is accessed; its value is cached in case it is needed again. The object may have any type, and different stream elements may have different types. If the `stream-car` is never accessed, the object stored there is never evaluated. Likewise, the `stream-cdr` is a promise to return a stream, and is only forced on demand.

#### 7.5.28.2 SRFI-41 Stream Primitives

This library provides eight operators: constructors for `stream-null` and `stream-pair`, type predicates for streams and the two kinds of streams, accessors for both fields of a `stream-pair`, and a lambda that creates procedures that return streams.

```scheme
stream-null  [Scheme Variable]
   A promise that, when forced, is a single object, distinguishable from all other objects, that represents the null stream. `stream-null` is immutable and unique.

stream-cons object-expr stream-expr  [Scheme Syntax]
   Creates a newly-allocated stream containing a promise that, when forced, is a `stream-pair` with object-expr in its `stream-car` and stream-expr in its `stream-cdr`. Neither object-expr nor stream-expr is evaluated when stream-cons is called.
   Once created, a `stream-pair` is immutable; there is no `stream-set-car!` or `stream-set-cdr!` that modifies an existing stream-pair. There is no dotted-pair or improper stream as with lists.
```
stream? object  [Scheme Procedure]
Returns true if object is a stream, otherwise returns false. If object is a stream, its promise will not be forced. If (stream? obj) returns true, then one of (stream-null? obj) or (stream-pair? obj) will return true and the other will return false.

stream-null? object  [Scheme Procedure]
Returns true if object is the distinguished null stream, otherwise returns false. If object is a stream, its promise will be forced.

stream-pair? object  [Scheme Procedure]
Returns true if object is a stream-pair constructed by stream-cons, otherwise returns false. If object is a stream, its promise will be forced.

stream-car stream  [Scheme Procedure]
Returns the object stored in the stream-car of stream. An error is signalled if the argument is not a stream-pair. This causes the object-expr passed to stream-cons to be evaluated if it had not yet been; the value is cached in case it is needed again.

stream-cdr stream  [Scheme Procedure]
Returns the stream stored in the stream-cdr of stream. An error is signalled if the argument is not a stream-pair.

stream-lambda formals body  . . .  [Scheme Syntax]
Creates a procedure that returns a promise to evaluate the body of the procedure. The last body expression to be evaluated must yield a stream. As with normal lambda, formals may be a single variable name, in which case all the formal arguments are collected into a single list, or a list of variable names, which may be null if there are no arguments, proper if there are an exact number of arguments, or dotted if a fixed number of arguments is to be followed by zero or more arguments collected into a list. Body must contain at least one expression, and may contain internal definitions preceding any expressions to be evaluated.

```
(define strm123
   (stream-cons 1
      (stream-cons 2
         (stream-cons 3
            stream-null))))

(stream-car strm123) ⇒ 1
(stream-car (stream-cdr strm123)) ⇒ 2

(stream-pair?
   (stream-cdr
      (stream-cons (/ 1 0) stream-null))) ⇒ #f

(stream? (list 1 2 3)) ⇒ #f

(define iter
   (stream-lambda (f x)
```

(stream-cons x (iter f (f x))))

(define nats (iter (lambda (x) (+ x 1)) 0))

(stream-car (stream-cdr nats)) ⇒ 1

(define stream-add
  (stream-lambda (s1 s2)
    (stream-cons
      (+ (stream-car s1) (stream-car s2))
      (stream-add (stream-cdr s1)
        (stream-cdr s2))))))

(define evens (stream-add nats nats))

(stream-car evens) ⇒ 0
(stream-car (stream-cdr evens)) ⇒ 2
(stream-car (stream-cdr (stream-cdr evens))) ⇒ 4

7.5.28.3 SRFI-41 Stream Library

define-stream (name args ... ) body ...

Creates a procedure that returns a stream, and may appear anywhere a normal define may appear, including as an internal definition. It may contain internal definitions of its own. The defined procedure takes arguments in the same way as stream-lambda. define-stream is syntactic sugar on stream-lambda; see also stream-let, which is also a sugaring of stream-lambda.

A simple version of stream-map that takes only a single input stream calls itself recursively:

(define-stream (stream-map proc strm)
  (if (stream-null? strm)
      stream-null
      (stream-cons
        (proc (stream-car strm))
        (stream-map proc (stream-cdr strm))))))

list->stream list

Returns a newly-allocated stream containing the elements from list.

port->stream [port]

Returns a newly-allocated stream containing in its elements the characters on the port. If port is not given it defaults to the current input port. The returned stream has finite length and is terminated by stream-null.

It looks like one use of port->stream would be this:

(define s ;wrong!
  (with-input-from-file filename
    (lambda () (port->stream))))
But that fails, because `with-input-from-file` is eager, and closes the input port prematurely, before the first character is read. To read a file into a stream, say:

```scheme
(define-stream (file->stream filename)
  (let ((p (open-input-file filename)))
    (stream-let loop ((c (read-char p)))
      (if (eof-object? c)
          (begin (close-input-port p)
                 stream-null)
          (stream-cons c
                      (loop (read-char p)))))))
```

**stream object-expr ...**  
[Scheme Syntax]
Creates a newly-allocated stream containing in its elements the objects, in order. The `object-expr`s are evaluated when they are accessed, not when the stream is created. If no objects are given, as in `(stream)`, the null stream is returned. See also `list->stream`.

```scheme
(define strm123 (stream 1 2 3))
```

; (/ 1 0) not evaluated when stream is created
```
(define s (stream 1 (/ 1 0) -1))
```

**stream->list [n] stream**  
[Scheme Procedure]
Returns a newly-allocated list containing in its elements the first `n` items in `stream`. If `stream` has less than `n` items, all the items in the stream will be included in the returned list. If `n` is not given it defaults to infinity, which means that unless `stream` is finite `stream->list` will never return.

```scheme
(stream->list 10
  (stream-map (lambda (x) (* x x))
              (stream-from 0)))
⇒ (0 1 4 9 16 25 36 49 64 81)
```

**stream-append stream ...**  
[Scheme Procedure]
Returns a newly-allocated stream containing in its elements those elements contained in its input `streams`, in order of input. If any of the input streams is infinite, no elements of any of the succeeding input streams will appear in the output stream. See also `stream-concat`.

**stream-concat stream**  
[Scheme Procedure]
Takes a `stream` consisting of one or more streams and returns a newly-allocated stream containing all the elements of the input streams. If any of the streams in the input `stream` is infinite, any remaining streams in the input stream will never appear in the output stream. See also `stream-append`.

**stream-constant object ...**  
[Scheme Procedure]
Returns a newly-allocated stream containing in its elements the `objects`, repeating in succession forever.

```
(stream-constant 1) ⇒ 1 1 1 ...
(stream-constant #t #f) ⇒ #t #f #t #f #t #f ...
```
**stream-drop** \( n \) stream

[Scheme Procedure]

Returns the suffix of the input stream that starts at the next element after the first \( n \) elements. The output stream shares structure with the input stream; thus, promises forced in one instance of the stream are also forced in the other instance of the stream. If the input stream has less than \( n \) elements, **stream-drop** returns the null stream. See also **stream-take**.

**stream-drop-while** \( \text{pred} \) stream

[Scheme Procedure]

Returns the suffix of the input stream that starts at the first element \( x \) for which \((\text{pred} \ x)\) returns false. The output stream shares structure with the input stream. See also **stream-take-while**.

**stream-filter** \( \text{pred} \) stream

[Scheme Procedure]

Returns a newly-allocated stream that contains only those elements \( x \) of the input stream which satisfy the predicate \( \text{pred} \).

\[
(\text{stream-filter} \ \text{odd?} \ (\text{stream-from} \ 0)) \Rightarrow 1 3 5 7 9 \ldots
\]

**stream-fold** \( \text{proc} \) base stream

[Scheme Procedure]

Applies a binary procedure \( \text{proc} \) to \( \text{base} \) and the first element of \( \text{stream} \) to compute a new \( \text{base} \), then applies the procedure to the new \( \text{base} \) and the next element of \( \text{stream} \) to compute a succeeding \( \text{base} \), and so on, accumulating a value that is finally returned as the value of **stream-fold** when the end of the stream is reached. \( \text{stream} \) must be finite, or **stream-fold** will enter an infinite loop. See also **stream-scan**, which is similar to **stream-fold**, but useful for infinite streams. For readers familiar with other functional languages, this is a left-fold; there is no corresponding right-fold, since right-fold relies on finite streams that are fully-evaluated, in which case they may as well be converted to a list.

**stream-for-each** \( \text{proc} \) stream . . .

[Scheme Procedure]

Applies \( \text{proc} \) element-wise to corresponding elements of the input streams for side-effects; it returns nothing. **stream-for-each** stops as soon as any of its input streams is exhausted.

**stream-from** first \([\text{step}]\)

[Scheme Procedure]

Creates a newly-allocated stream that contains first as its first element and increments each succeeding element by step. If step is not given it defaults to 1. first and step may be of any numeric type. **stream-from** is frequently useful as a generator in **stream-of** expressions. See also **stream-range** for a similar procedure that creates finite streams.

**stream-iterate** \( \text{proc} \) base

[Scheme Procedure]

Creates a newly-allocated stream containing base in its first element and applies \( \text{proc} \) to each element in turn to determine the succeeding element. See also **stream-unfold** and **stream-unfolds**.

**stream-length** stream

[Scheme Procedure]

Returns the number of elements in the stream; it does not evaluate its elements. **stream-length** may only be used on finite streams; it enters an infinite loop with infinite streams.
stream-let tag ((var expr) . . .) body . . .  

Creates a local scope that binds each variable to the value of its corresponding expression. It additionally binds tag to a procedure which takes the bound variables as arguments and body as its defining expressions, binding the tag with stream-lambda.  

stream-let is in scope within body, and may be called recursively. When the expanded expression defined by the stream-let is evaluated, stream-let evaluates the expressions in its body in an environment containing the newly-bound variables, returning the value of the last expression evaluated, which must yield a stream.  

stream-let provides syntactic sugar on stream-lambda, in the same manner as normal let provides syntactic sugar on normal lambda. However, unlike normal let, the tag is required, not optional, because unnamed stream-let is meaningless.  

For example, stream-member returns the first stream-pair of the input strm with a stream-car x that satisfies (eql? obj x), or the null stream if x is not present in strm.

(define-stream (stream-member eql? obj strm)
  (stream-let loop ((strm strm))
    (cond ((stream-null? strm) strm)
          ((eql? obj (stream-car strm)) strm)
          (else (loop (stream-cdr strm))))))

stream-map proc stream . . .  

Applies proc element-wise to corresponding elements of the input streams, returning a newly-allocated stream containing elements that are the results of those procedure applications. The output stream has as many elements as the minimum-length input stream, and may be infinite.

stream-match stream clause . . .  

Provides pattern-matching for streams. The input stream is an expression that evaluates to a stream. Clauses are of the form (pattern [fender] expression), consisting of a pattern that matches a stream of a particular shape, an optional fender that must succeed if the pattern is to match, and an expression that is evaluated if the pattern matches. There are four types of patterns:

- () matches the null stream.
- (pat0 pat1 . . .) matches a finite stream with length exactly equal to the number of pattern elements.
- (pat0 pat1 . . . pat-rest) matches an infinite stream, or a finite stream with length at least as great as the number of pattern elements before the literal dot.
- pat matches an entire stream. Should always appear last in the list of clauses; it’s not an error to appear elsewhere, but subsequent clauses could never match.

Each pattern element may be either:

- An identifier, which matches any stream element. Additionally, the value of the stream element is bound to the variable named by the identifier, which is in scope in the fender and expression of the corresponding clause. Each identifier in a single pattern must be unique.
- A literal underscore (_), which matches any stream element but creates no bindings.
The patterns are tested in order, left-to-right, until a matching pattern is found; if \textit{fender} is present, it must evaluate to a true value for the match to be successful. Pattern variables are bound in the corresponding \textit{fender} and \textit{expression}. Once the matching pattern is found, the corresponding \textit{expression} is evaluated and returned as the result of the match. An error is signaled if no pattern matches the input \textit{stream}.

\texttt{stream-match} is often used to distinguish null streams from non-null streams, binding \textit{head} and \textit{tail}:

\begin{verbatim}
(define (len strm)
  (stream-match strm
    (() 0)
    ((head . tail) (+ 1 (len tail)))))
\end{verbatim}

Fenders can test the common case where two stream elements must be identical; the \textit{else} pattern is an identifier bound to the entire stream, not a keyword as in \texttt{cond}.

\begin{verbatim}
(stream-match strm
  ((x y . _) (equal? x y) 'ok)
  (else 'error))
\end{verbatim}

A more complex example uses two nested matchers to match two different stream arguments; \texttt{(stream-merge lt? . strms)} stably merges two or more streams ordered by the \textit{lt?} predicate:

\begin{verbatim}
(define-stream (stream-merge lt? . strms)
  (define-stream (merge xx yy)
    (stream-match xx (() yy) ((x . xs)
      (stream-match yy (() xx) ((y . ys)
        (if (lt? y x)
          (stream-cons y (merge xx ys))
          (stream-cons x (merge xs yy)))))))))
  (stream-let loop ((strms strms))
    (cond ((null? strms) stream-null)
      ((null? (cdr strms)) (car strms))
      (else (merge (car strms)
        (apply stream-merge lt? (cdr strms)))))))
\end{verbatim}

\texttt{stream-of expr clause . . .} \hspace{1cm} [Scheme Syntax]

Provides the syntax of stream comprehensions, which generate streams by means of looping expressions. The result is a stream of objects of the type returned by \textit{expr}. There are four types of clauses:

- (\textit{var in stream-expr}) loops over the elements of \textit{stream-expr}, in order from the start of the stream, binding each element of the stream in turn to \textit{var}. \texttt{stream-from} and \texttt{stream-range} are frequently useful as generators for \textit{stream-expr}.

- (\textit{var is expr}) binds \textit{var} to the value obtained by evaluating \textit{expr}.

- (\textit{pred expr}) includes in the output stream only those elements \textit{x} which satisfy the predicate \textit{pred}. 


The scope of variables bound in the stream comprehension is the clauses to the right of the binding clause (but not the binding clause itself) plus the result expression. When two or more generators are present, the loops are processed as if they are nested from left to right; that is, the rightmost generator varies fastest. A consequence of this is that only the first generator may be infinite and all subsequent generators must be finite. If no generators are present, the result of a stream comprehension is a stream containing the result expression; thus, `(stream-of 1)` produces a finite stream containing only the element 1.

```
(stream-of (* x x)
  (x in (stream-range 0 10))
  (even? x))
⇒ 0 4 16 36 64
```

```
(stream-of (list a b)
  (a in (stream-range 1 4))
  (b in (stream-range 1 3)))
⇒ (1 1) (1 2) (2 1) (2 2) (3 1) (3 2)
```

```
(stream-of (list i j)
  (i in (stream-range 1 5))
  (j in (stream-range (+ i 1) 5)))
⇒ (1 2) (1 3) (1 4) (2 3) (2 4) (3 4)
```

```scheme
(stream-range first past [step])
```

Returns a newly-allocated stream that contains `first` as its first element and increments each succeeding element by `step`. The stream is finite and ends before `past`, which is not an element of the stream. If `step` is not given it defaults to 1 if `first` is less than `past` and -1 otherwise. `first`, `past` and `step` may be of any real numeric type. `stream-range` is frequently useful as a generator in `stream-of` expressions. See also `stream-from` for a similar procedure that creates infinite streams.

```
(stream-range 0 10) ⇒ 0 1 2 3 4 5 6 7 8 9
(stream-range 0 10 2) ⇒ 0 2 4 6 8
```

Successive elements of the stream are calculated by adding `step` to `first`, so if any of `first`, `past` or `step` are inexact, the length of the output stream may differ from

```
(ceiling (- (/ (- past first) step) 1))
```

```scheme
(stream-ref stream n)
```

Returns the `n`th element of `stream`, counting from zero. An error is signaled if `n` is greater than or equal to the length of `stream`.

```
(define (fact n)
  (stream-ref
    (stream-scan * 1 (stream-from 1))
    n))
```

```scheme
(stream-reverse stream)
```

Returns a newly-allocated stream containing the elements of the input `stream` but in reverse order. `stream-reverse` may only be used with finite streams; it enters an
infinite loop with infinite streams. \texttt{stream-reverse} does not force evaluation of the elements of the stream.

\texttt{stream-scan proc base stream} \hspace{1cm} \text{[Scheme Procedure]}

Accumulates the partial folds of an input \texttt{stream} into a newly-allocated output stream. The output stream is the \texttt{base} followed by \((\texttt{stream-fold proc base (stream-take i stream)})\) for each of the first \(i\) elements of \texttt{stream}.

\[
\begin{align*}
(\texttt{stream-scan } + 0 \texttt{ (stream-from 1)}) & \Rightarrow (\texttt{stream } 0 \hspace{1em} 1 \hspace{1em} 3 \hspace{1em} 6 \hspace{1em} 10 \hspace{1em} 15 \ldots) \\
(\texttt{stream-scan } * 1 \texttt{ (stream-from 1)}) & \Rightarrow (\texttt{stream } 1 \hspace{1em} 1 \hspace{1em} 2 \hspace{1em} 6 \hspace{1em} 24 \hspace{1em} 120 \ldots)
\end{align*}
\]

\texttt{stream-take n stream} \hspace{1cm} \text{[Scheme Procedure]}

Returns a newly-allocated stream containing the first \(n\) elements of the input \texttt{stream}. If the input \texttt{stream} has less than \(n\) elements, so does the output stream. See also \texttt{stream-drop}.

\texttt{stream-take-while pred stream} \hspace{1cm} \text{[Scheme Procedure]}

Takes a predicate and a \texttt{stream} and returns a newly-allocated stream containing those elements \(x\) that form the maximal prefix of the input stream which satisfy \texttt{pred}. See also \texttt{stream-drop-while}.

\texttt{stream-unfold map pred gen base} \hspace{1cm} \text{[Scheme Procedure]}

The fundamental recursive stream constructor. It constructs a stream by repeatedly applying \texttt{gen} to successive values of \texttt{base}, in the manner of \texttt{stream-iterate}, then applying \texttt{map} to each of the values so generated, appending each of the mapped values to the output stream as long as \((\texttt{pred? base})\) returns a true value. See also \texttt{stream-iterate} and \texttt{stream-unfolds}.

The expression below creates the finite stream ‘0 1 4 9 16 25 36 49 64 81’. Initially the \texttt{base} is 0, which is less than 10, so \texttt{map} squares the \texttt{base} and the mapped value becomes the first element of the output stream. Then \texttt{gen} increments the \texttt{base} by 1, so it becomes 1; this is less than 10, so \texttt{map} squares the new \texttt{base} and 1 becomes the second element of the output stream. And so on, until the base becomes 10, when \texttt{pred} stops the recursion and \texttt{stream-null} ends the output stream.

\[
\begin{align*}
(\texttt{stream-unfold} \\
(\lambda (x) \ (\texttt{expt } x \ 2)) ; \texttt{map} \\
(\lambda (x) \ (< \ x \ 10)) \ ; \texttt{pred?} \\
(\lambda (x) \ (+ \ x \ 1)) \ ; \texttt{gen} \\
0) \ ; \texttt{base}
\end{align*}
\]

\texttt{stream-unfolds proc seed} \hspace{1cm} \text{[Scheme Procedure]}

Returns \(n\) newly-allocated streams containing those elements produced by successive calls to the generator \texttt{proc}, which takes the current \texttt{seed} as its argument and returns \(n+1\) values

\[
(\texttt{proc seed}) \Rightarrow \texttt{seed} \ \texttt{result}_0 \ldots \ \texttt{result}_\texttt{n-1}
\]
where the returned seed is the input seed to the next call to the generator and result_{i} indicates how to produce the next element of the i\text{th} result stream:

- **(value):** value is the next car of the result stream.
- **#f:** no value produced by this iteration of the generator proc for the result stream.
- **():** the end of the result stream.

It may require multiple calls of proc to produce the next element of any particular result stream. See also stream-iterate and stream-unfold.

```
(define (stream-partition pred? strm)
  (stream-unfolds
   (lambda (s)
     (if (stream-null? s)
         (values s '() '())
         (let ((a (stream-car s))
               (d (stream-cdr s)))
           (if (pred? a)
               (values d (list a) #f)
               (values d #f (list a))))))
   strm))

(call-with-values
 (lambda ()
   (stream-partition odd? (stream-range 1 6)))
 (lambda (odds evens)
   (list (stream->list odds)
         (stream->list evens))))
⇒ ((1 3 5) (2 4))
```

**stream-zip stream ...**

Returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input streams. The output stream is as long as the shortest input stream, if any of the input streams is finite, or is infinite if all the input streams are infinite.

### 7.5.29 SRFI-42 - Eager Comprehensions

See the specification of SRFI-42 ([http://srfi.schemers.org/srfi-42/srfi-42.html](http://srfi.schemers.org/srfi-42/srfi-42.html)).

### 7.5.30 SRFI-43 - Vector Library


SRFI-43 implements a comprehensive library of vector operations. It can be made available with:

```
(use-modules (srfi srfi-43))
```
7.5.30.1 SRFI-43 Constructors

**make-vector size [fill]**

Create and return a vector of size `size`, optionally filling it with `fill`. The default value of `fill` is unspecified.

\[
\text{(make-vector 5 3)} \Rightarrow \text{#(3 3 3 3 3)}
\]

**vector x . . .**

Create and return a vector whose elements are `x` . . ..

\[
\text{(vector 0 1 2 3 4)} \Rightarrow \text{#(0 1 2 3 4)}
\]

**vector-unfold \( f \) length initial-seed . . .**

The fundamental vector constructor. Create a vector whose length is `length` and iterates across each index `k` from 0 up to `length - 1`, applying `f` at each iteration to the current index and current seeds, in that order, to receive `n + 1` values: the element to put in the `kth` slot of the new vector, and `n` new seeds for the next iteration. It is an error for the number of seeds to vary between iterations.

\[
\text{(vector-unfold (lambda (i x) (values x (- x 1))) 10 0)} \\
\Rightarrow \text{#(0 -1 -2 -3 -4 -5 -6 -7 -8 -9)}
\]

\[
\text{(vector-unfold values 10)} \\
\Rightarrow \text{#(0 1 2 3 4 5 6 7 8 9)}
\]

**vector-unfold-right \( f \) length initial-seed . . .**

Like `vector-unfold`, but it uses `f` to generate elements from right-to-left, rather than left-to-right.

\[
\text{(vector-unfold-right (lambda (i x) (values x (+ x 1))) 10 0)} \\
\Rightarrow \text{#(9 8 7 6 5 4 3 2 1 0)}
\]

**vector-copy vec [start [end [fill]]]**

Allocate a new vector whose length is `end - start` and fills it with elements from `vec`, taking elements from `vec` starting at index `start` and stopping at index `end`. `start` defaults to 0 and `end` defaults to the value of `(vector-length vec)`. If `end` extends beyond the length of `vec`, the slots in the new vector that obviously cannot be filled by elements from `vec` are filled with `fill`, whose default value is unspecified.

\[
\text{(vector-copy ’#(a b c d e f g h i))} \\
\Rightarrow \text{#(a b c d e f g h i)}
\]

\[
\text{(vector-copy ’#(a b c d e f g h i) 6)} \\
\Rightarrow \text{#(g h i)}
\]

\[
\text{(vector-copy ’#(a b c d e f g h i) 3 6)} \\
\Rightarrow \text{#(d e f)}
\]

\[
\text{(vector-copy ’#(a b c d e f g h i) 6 12 ’x)} \\
\Rightarrow \text{#(g h i x x x)}
\]
**vector-reverse-copy vec [start [end]]**  
[Scheme Procedure]  
Like vector-copy, but it copies the elements in the reverse order from vec.

```scheme
(vector-reverse-copy '#(5 4 3 2 1 0) 1 5)
⇒ #(1 2 3 4)
```

**vector-append vec ...**  
[Scheme Procedure]  
Return a newly allocated vector that contains all elements in order from the subsequent locations in vec . . .

```scheme
(vector-append '#(a) '#(b c d))
⇒ #(a b c d)
```

**vector-concatenate list-of-vectors**  
[Scheme Procedure]  
Append each vector in list-of-vectors. Equivalent to (apply vector-append list-of-vectors).

```scheme
(vector-concatenate '(#(a b) #(c d)))
⇒ #(a b c d)
```

### 7.5.30.2 SRFI-43 Predicates

**vector? obj**  
[Scheme Procedure]  
Return true if obj is a vector, else return false.

**vector-empty? vec**  
[Scheme Procedure]  
Return true if vec is empty, i.e. its length is 0, else return false.

**vector= elt=? vec ...**  
[Scheme Procedure]  
Return true if the vectors vec ... have equal lengths and equal elements according to elt=? . elt=? is always applied to two arguments. Element comparison must be consistent with eq? in the following sense: if (eq? a b) returns true, then (elt=? a b) must also return true. The order in which comparisons are performed is unspecified.

### 7.5.30.3 SRFI-43 Selectors

**vector-ref vec i**  
[Scheme Procedure]  
Return the element at index i in vec. Indexing is based on zero.

**vector-length vec**  
[Scheme Procedure]  
Return the length of vec.

### 7.5.30.4 SRFI-43 Iteration

**vector-fold kons knil vec1 vec2 ...**  
[Scheme Procedure]  
The fundamental vector iterator. kons is iterated over each index in all of the vectors, stopping at the end of the shortest; kons is applied as

```scheme
(kons i state (vector-ref vec1 i) (vector-ref vec2 i) ...)
```

where state is the current state value, and i is the current index. The current state value begins with knil, and becomes whatever kons returned at the respective iteration. The iteration is strictly left-to-right.
**vector-fold-right** **kons knil vec1 vec2** ...  
[Scheme Procedure]  
Similar to **vector-fold**, but it iterates right-to-left instead of left-to-right.

**vector-map** **f vec1 vec2** ...  
[Scheme Procedure]  
Return a new vector of the shortest size of the vector arguments. Each element at index `i` of the new vector is mapped from the old vectors by

\[(f \ i \ (vector-ref \ vec1 \ i) \ (vector-ref \ vec2 \ i) \ ...)\]

The dynamic order of application of `f` is unspecified.

**vector-map!** **f vec1 vec2** ...  
[Scheme Procedure]  
Similar to **vector-map**, but rather than mapping the new elements into a new vector, the new mapped elements are destructively inserted into `vec1`. The dynamic order of application of `f` is unspecified.

**vector-for-each** **f vec1 vec2** ...  
[Scheme Procedure]  
Call `(f \ i \ (vector-ref \ vec1 \ i) \ (vector-ref \ vec2 \ i) \ ...)` for each index `i` less than the length of the shortest vector passed. The iteration is strictly left-to-right.

**vector-count** **pred? vec1 vec2** ...  
[Scheme Procedure]  
Count the number of parallel elements in the vectors that satisfy `pred?`, which is applied, for each index `i` less than the length of the smallest vector, to `i` and each parallel element in the vectors at that index, in order.

\[(vector-count \ (lambda \ (i \ elt) \ (even? \ elt)) \ ' (3 1 4 1 5 9 2 5 6))\]

⇒ 3

\[(vector-count \ (lambda \ (i \ x \ y) \ (< \ x \ y)) \ ' (1 3 6 9) \ ' (2 4 6 8 10 12))\]

⇒ 2

### 7.5.30.5 SRFI-43 Searching

**vector-index** **pred? vec1 vec2** ...  
[Scheme Procedure]  
Find and return the index of the first elements in `vec1 vec2` ... that satisfy `pred?`. If no matching element is found by the end of the shortest vector, return `#f`.

\[(vector-index \ even? \ ' (3 1 4 1 5 9))\]

⇒ 2

\[(vector-index \ < \ ' (3 1 4 1 5 9 2 5 6) \ ' (2 7 1 8 2))\]

⇒ 1

\[(vector-index = \ ' (3 1 4 1 5 9 2 5 6) \ ' (2 7 1 8 2))\]

⇒ `#f`

**vector-index-right** **pred? vec1 vec2** ...  
[Scheme Procedure]  
Like **vector-index**, but it searches right-to-left, rather than left-to-right. Note that the SRFI 43 specification requires that all the vectors must have the same length, but both the SRFI 43 reference implementation and Guile’s implementation allow vectors with unequal lengths, and start searching from the last index of the shortest vector.
vector-skip pred? vec1 vec2 . . .  
[Scheme Procedure]
Find and return the index of the first elements in vec1 vec2 . . . that do not satisfy pred?. If no matching element is found by the end of the shortest vector, return #f. Equivalent to vector-index but with the predicate inverted.

(vector-skip number? '#(1 2 a b 3 4 c d)) ⇒ 2

vector-skip-right pred? vec1 vec2 . . .  
[Scheme Procedure]
Like vector-skip, but it searches for a non-matching element right-to-left, rather than left-to-right. Note that the SRFI 43 specification requires that all the vectors must have the same length, but both the SRFI 43 reference implementation and Guile's implementation allow vectors with unequal lengths, and start searching from the last index of the shortest vector.

(vector-binary-search vec value cmp [start [end]])  
[Scheme Procedure]
Find and return an index of vec between start and end whose value is value using a binary search. If no matching element is found, return #f. The default start is 0 and the default end is the length of vec.

cmp must be a procedure of two arguments such that (cmp a b) returns a negative integer if a<b, a positive integer if a>b, or zero if a=b. The elements of vec must be sorted in non-decreasing order according to cmp.

Note that SRFI 43 does not document the start and end arguments, but both its reference implementation and Guile’s implementation support them.

(define (char-cmp c1 c2)
  (cond ((char<? c1 c2) -1)
        ((char>? c1 c2) 1)
        (else 0)))

(vector-binary-search '#(#\a #\b #\c #\d #\e #\f #\g #\h)
                      #\g
                    char-cmp)
⇒ 6

vector-any pred? vec1 vec2 . . .  
[Scheme Procedure]
Find the first parallel set of elements from vec1 vec2 . . . for which pred? returns a true value. If such a parallel set of elements exists, vector-any returns the value that pred? returned for that set of elements. The iteration is strictly left-to-right.

vector-every pred? vec1 vec2 . . .  
[Scheme Procedure]
If, for every index i between 0 and the length of the shortest vector argument, the set of elements (vector-ref vec1 i) (vector-ref vec2 i) . . . satisfies pred?, vector-every returns the value that pred? returned for the last set of elements, at the last index of the shortest vector. Otherwise it returns #f. The iteration is strictly left-to-right.

7.5.30.6 SRFI-43 Mutators

vector-set! vec i value  
[Scheme Procedure]
Assign the contents of the location at i in vec to value.
vector-swap! vec i j  
Swap the values of the locations in vec at i and j.

vector-fill! vec fill [start [end]]  
Assign the value of every location in vec between start and end to fill. start defaults to 0 and end defaults to the length of vec.

vector-reverse! vec [start [end]]  
Destructively reverse the contents of vec between start and end. start defaults to 0 and end defaults to the length of vec.

vector-copy! target tstart source [sstart [send]]  
Copy a block of elements from source to target, both of which must be vectors, starting in target at tstart and starting in source at sstart, ending when (send - sstart) elements have been copied. It is an error for target to have a length less than (tstart + send - sstart). sstart defaults to 0 and send defaults to the length of source.

vector-reverse-copy! target tstart source [sstart [send]]  
Like vector-copy!, but this copies the elements in the reverse order. It is an error if target and source are identical vectors and the target and source ranges overlap; however, if tstart = sstart, vector-reverse-copy! behaves as (vector-reverse! target tstart send) would.

7.5.30.7 SRFI-43 Conversion

vector->list vec [start [end]]  
Return a newly allocated list containing the elements in vec between start and end. start defaults to 0 and end defaults to the length of vec.

reverse-vector->list vec [start [end]]  
Like vector->list, but the resulting list contains the specified range of elements of vec in reverse order.

list->vector proper-list [start [end]]  
Return a newly allocated vector of the elements from proper-list with indices between start and end. start defaults to 0 and end defaults to the length of proper-list. Note that SRFI 43 does not document the start and end arguments, but both its reference implementation and Guile’s implementation support them.

reverse-list->vector proper-list [start [end]]  
Like list->vector, but the resulting vector contains the specified range of elements of proper-list in reverse order. Note that SRFI 43 does not document the start and end arguments, but both its reference implementation and Guile’s implementation support them.

7.5.31 SRFI-45 - Primitives for Expressing Iterative Lazy Algorithms

This subsection is based on the specification of SRFI-45 (http://srfi.schemers.org/srfi-45/srfi-45.html) written by André van Tonder.
Lazy evaluation is traditionally simulated in Scheme using `delay` and `force`. However, these primitives are not powerful enough to express a large class of lazy algorithms that are iterative. Indeed, it is folklore in the Scheme community that typical iterative lazy algorithms written using delay and force will often require unbounded memory.

This SRFI provides set of three operations: \{`lazy`, `delay`, `force`\}, which allow the programmer to succinctly express lazy algorithms while retaining bounded space behavior in cases that are properly tail-recursive. A general recipe for using these primitives is provided. An additional procedure `eager` is provided for the construction of eager promises in cases where efficiency is a concern.

Although this SRFI redefines `delay` and `force`, the extension is conservative in the sense that the semantics of the subset \{`delay`, `force`\} in isolation (i.e., as long as the program does not use `lazy`) agrees with that in R5RS. In other words, no program that uses the R5RS definitions of delay and force will break if those definition are replaced by the SRFI-45 definitions of delay and force.

Guile also adds `promise?` to the list of exports, which is not part of the official SRFI-45.

**promise? obj**  
[Scheme Procedure]  
Return true if `obj` is an SRFI-45 promise, otherwise return false.

**delay expression**  
[Scheme Syntax]  
Takes an expression of arbitrary type `a` and returns a promise of type `(Promise a)` which at some point in the future may be asked (by the `force` procedure) to evaluate the expression and deliver the resulting value.

**lazy expression**  
[Scheme Syntax]  
Takes an expression of type `(Promise a)` and returns a promise of type `(Promise a)` which at some point in the future may be asked (by the `force` procedure) to evaluate the expression and deliver the resulting promise.

**force expression**  
[Scheme Procedure]  
Takes an argument of type `(Promise a)` and returns a value of type `a` as follows: If a value of type `a` has been computed for the promise, this value is returned. Otherwise, the promise is first evaluated, then overwritten by the obtained promise or value, and then force is again applied (iteratively) to the promise.

**eager expression**  
[Scheme Procedure]  
Takes an argument of type `a` and returns a value of type `(Promise a)`. As opposed to `delay`, the argument is evaluated eagerly. Semantically, writing `(eager expression)` is equivalent to writing

\[
(\text{let } ((\text{value } expression)) (\text{delay value})).
\]

However, the former is more efficient since it does not require unnecessary creation and evaluation of thunks. We also have the equivalence

\[
(\text{delay expression}) = (\text{lazy } (\text{eager expression})).
\]

The following reduction rules may be helpful for reasoning about these primitives. However, they do not express the memoization and memory usage semantics specified above:

\[
\begin{align*}
(\text{force } (\text{delay expression})) & \rightarrow \text{expression} \\
(\text{force } (\text{lazy } \text{expression})) & \rightarrow (\text{force } \text{expression}) \\
(\text{force } \text{eager } \text{value}) & \rightarrow \text{value}
\end{align*}
\]
Correct usage

We now provide a general recipe for using the primitives \{lazy, delay, force\} to express lazy algorithms in Scheme. The transformation is best described by way of an example: Consider the stream-filter algorithm, expressed in a hypothetical lazy language as

\[
\text{(define (stream-filter p? s)}
\text{  (if (null? s) '())
      (let ((h (car s))
            (t (cdr s)))
        (if (p? h)
            (cons h (stream-filter p? t))
            (stream-filter p? t))))
\]

This algorithm can be expressed as follows in Scheme:

\[
\text{(define (stream-filter p? s)}
\text{  (lazy
      (if (null? (force s)) (delay '())
        (let ((h (car (force s)))
            (t (cdr (force s)))
            (if (p? h)
                (delay (cons h (stream-filter p? t)))
                (stream-filter p? t))))))}
\]

In other words, we

- wrap all constructors (e.g., '(), cons) with delay,
- apply force to arguments of deconstructors (e.g., car, cdr and null?),
- wrap procedure bodies with (lazy ...).

7.5.32 SRFI-46 Basic syntax-rules Extensions

Guile’s core syntax-rules supports the extensions specified by SRFI-46/R7RS. Tail patterns have been supported since at least Guile 2.0, and custom ellipsis identifiers have been supported since Guile 2.0.10. See Section 6.10.2 [Syntax Rules], page 262.

7.5.33 SRFI-55 - Requiring Features

SRFI-55 provides require-extension which is a portable mechanism to load selected SRFI modules. This is implemented in the Guile core, there’s no module needed to get SRFI-55 itself.

require-extension \textbf{clause1 clause2 ...} [library syntax]

Require the features of \textbf{clause1 clause2 ...}, throwing an error if any are unavailable.

A \textbf{clause} is of the form (identifier arg...). The only identifier currently supported is srfi and the arguments are SRFI numbers. For example to get SRFI-1 and SRFI-6,

\[
\text{(require-extension (srfi 1 6))}
\]

require-extension can only be used at the top-level.

A Guile-specific program can simply use-modules to load SRFIs not already in the core, require-extension is for programs designed to be portable to other Scheme implementations.
7.5.34 SRFI-60 - Integers as Bits

This SRFI provides various functions for treating integers as bits and for bitwise manipulations. These functions can be obtained with,

```
(use-modules (srfi srfi-60))
```

Integers are treated as infinite precision twos-complement, the same as in the core logical functions (see Section 6.6.2.13 [Bitwise Operations], page 125). And likewise bit indexes start from 0 for the least significant bit. The following functions in this SRFI are already in the Guile core,

- `logand`
- `logior`
- `logxor`
- `lognot`
- `logtest`
- `logcount`
- `integer-length`
- `logbit?`
- `ash`

```
(bitwise-and n1 ...
(bitwise-ior n1 ...
(bitwise-xor n1 ...
(bitwise-not n
(any-bits-set? j k
(bit-set? index n
(arithmetic-shift n count
(bit-field n start end
(bit-count n

Aliases for `logand`, `logior`, `logxor`, `lognot`, `logtest`, `logbit?`, `ash`, `bit-extract` respectively.

Note that the name `bit-count` conflicts with `bit-count` in the core (see Section 6.6.11 [Bit Vectors], page 191).

```
(bitwise-if mask n1 n0
(bitwise-merge mask n1 n0
```

Return an integer with bits selected from `n1` and `n0` according to `mask`. Those bits where `mask` has 1s are taken from `n1`, and those where `mask` has 0s are taken from `n0`.

```
(bitwise-if 3 #b0101 #b1010) ⇒ 9
```

```
log2-binary-factors n
first-set-bit n
```

Return a count of how many factors of 2 are present in `n`. This is also the bit index of the lowest 1 bit in `n`. If `n` is 0, the return is −1.

```
(log2-binary-factors 6) ⇒ 1
(log2-binary-factors -8) ⇒ 3
```

```
copy-bit index n newbit
```

Return `n` with the bit at `index` set according to `newbit`. `newbit` should be #t to set the bit to 1, or #f to set it to 0. Bits other than at `index` are unchanged in the return.

```
(copy-bit 1 #b0101 #t) ⇒ 7
```

```
copy-bit-field n newbits start end
```

Return `n` with the bits from `start` (inclusive) to `end` (exclusive) changed to the value `newbits`. 
The least significant bit in `newbits` goes to `start`, the next to `start + 1`, etc. Anything in `newbits` past the `end` given is ignored.

```
(copy-bit-field #b10000 #b11 1 3) ⇒ #b10110
```

**rotate-bit-field** `n count start end`

Return `n` with the bit field from `start` (inclusive) to `end` (exclusive) rotated upwards by `count` bits.

`count` can be positive or negative, and it can be more than the field width (it’ll be reduced modulo the width).

```
(rotate-bit-field #b10110 2 1 4) ⇒ #b1010
```

**reverse-bit-field** `n start end`

Return `n` with the bits from `start` (inclusive) to `end` (exclusive) reversed.

```
(reverse-bit-field #b101001 2 4) ⇒ #b100101
```

**integer->list** `n [len]`

Return bits from `n` in the form of a list of #t for 1 and #f for 0. The least significant `len` bits are returned, and the first list element is the most significant of those bits. If `len` is not given, the default is `(integer-length n)` (see Section 6.6.2.13 [Bitwise Operations], page 125).

```
(integer->list 6) ⇒ (#t #t #f)
(integer->list 1 4) ⇒ (#f #f #f #t)
```

**list->integer** `lst` [Function]

Return an integer formed bitwise from the given `lst` list of bools, or for `booleans->integer` from the `bool` arguments.

Each boolean is #t for a 1 and #f for a 0. The first element becomes the most significant bit in the return.

```
(list->integer '(#t #f #t #f)) ⇒ 10
```

### 7.5.35 SRFI-61 - A more general cond clause

This SRFI extends RnRS `cond` to support test expressions that return multiple values, as well as arbitrary definitions of test success. SRFI 61 is implemented in the Guile core; there’s no module needed to get SRFI-61 itself. Extended `cond` is documented in Section 6.13.2 [Simple Conditional Evaluation], page 299.

### 7.5.36 SRFI-62 - S-expression comments.

Starting from version 2.0, Guile’s `read` supports SRFI-62/R7RS S-expression comments by default.

### 7.5.37 SRFI-64 - A Scheme API for test suites.

See the specification of SRFI-64 (http://srfi.schemers.org/srfi-64/srfi-64.html).

### 7.5.38 SRFI-67 - Compare procedures

7.5.39 SRFI-69 - Basic hash tables

This is a portable wrapper around Guile’s built-in hash table and weak table support. See Section 6.6.22 [Hash Tables], page 237, for information on that built-in support. Above that, this hash-table interface provides association of equality and hash functions with tables at creation time, so variants of each function are not required, as well as a procedure that takes care of most uses for Guile hash table handles, which this SRFI does not provide as such.

Access it with:

(use-modules (srfi srfi-69))

7.5.39.1 Creating hash tables

make-hash-table [equal-proc hash-proc #:weak weakness] [start-size]  
(Scheme Procedure)

Create and answer a new hash table with equal-proc as the equality function and hash-proc as the hashing function.

By default, equal-proc is equal?. It can be any two-argument procedure, and should answer whether two keys are the same for this table’s purposes.

My default hash-proc assumes that equal-proc is no coarser than equal? unless it is literally string-ci=? . If provided, hash-proc should be a two-argument procedure that takes a key and the current table size, and answers a reasonably good hash integer between 0 (inclusive) and the size (exclusive).

weakness should be #f or a symbol indicating how “weak” the hash table is:

#f An ordinary non-weak hash table. This is the default.
key When the key has no more non-weak references at GC, remove that entry.
value When the value has no more non-weak references at GC, remove that entry.
key-or-value When either has no more non-weak references at GC, remove the association.

As a legacy of the time when Guile couldn’t grow hash tables, start-size is an optional integer argument that specifies the approximate starting size for the hash table, which will be rounded to an algorithmically-sounder number.

By coarser than equal?, we mean that for all x and y values where (equal-proc x y), (equal? x y) as well. If that does not hold for your equal-proc, you must provide a hash-proc.

In the case of weak tables, remember that references above always refers to eq?-wise references. Just because you have a reference to some string "foo" doesn’t mean that an association with key "foo" in a weak-key table won’t be collected; it only counts as a reference if the two "foo"s are eq?, regardless of equal-proc. As such, it is usually only sensible to use eq? and hashq as the equivalence and hash functions for a weak table. See Section 6.19.3 [Weak References], page 404, for more information on Guile’s built-in weak table support.
alist->hash-table  alist [equal-proc hash-proc #:weak weakness start-size]  [Scheme Procedure]
As with make-hash-table, but initialize it with the associations in alist. Where keys are repeated in alist, the leftmost association takes precedence.

7.5.39.2 Accessing table items

hash-table-ref  table key [default-thunk]  [Scheme Procedure]
hash-table-ref/default  table key default  [Scheme Procedure]
Answer the value associated with key in table. If key is not present, answer the result of invoking the thunk default-thunk, which signals an error instead by default.
hash-table-ref/default is a variant that requires a third argument, default, and answers default itself instead of invoking it.

hash-table-set!  table key new-value  [Scheme Procedure]
Set key to new-value in table.

hash-table-delete!  table key  [Scheme Procedure]
Remove the association of key in table, if present. If absent, do nothing.

hash-table-exists?  table key  [Scheme Procedure]
Answer whether key has an association in table.

hash-table-update!  table key modifier [default-thunk]  [Scheme Procedure]
hash-table-update!/default  table key modifier default  [Scheme Procedure]
Replace key’s associated value in table by invoking modifier with one argument, the old value.
If key is not present, and default-thunk is provided, invoke it with no arguments to get the “old value” to be passed to modifier as above. If default-thunk is not provided in such a case, signal an error.
hash-table-update!/default is a variant that requires the fourth argument, which is used directly as the “old value” rather than as a thunk to be invoked to retrieve the “old value”.

7.5.39.3 Table properties

hash-table-size  table  [Scheme Procedure]
Answer the number of associations in table. This is guaranteed to run in constant time for non-weak tables.

hash-table-keys  table  [Scheme Procedure]
Answer an unordered list of the keys in table.

hash-table-values  table  [Scheme Procedure]
Answer an unordered list of the values in table.

hash-table-walk  table proc  [Scheme Procedure]
Invoke proc once for each association in table, passing the key and value as arguments.
hash-table-fold  

```
invoke (proc key value previous) for each key and value in table, where previous is the result of the previous invocation, using init as the first previous value. Answer the final proc result.
```

hash-table->alist  

```
Answer an alist where each association in table is an association in the result.
```

### 7.5.39.4 Hash table algorithms

Each hash table carries an **equivalence function** and a **hash function**, used to implement key lookups. Beginning users should follow the rules for consistency of the default hash-proc specified above. Advanced users can use these to implement their own equivalence and hash functions for specialized lookup semantics.

- **hash-table-equivalence-function**  
  
```scheme```
  hash-table
  ```
  Answer the equivalence and hash function of hash-table, respectively.
```

- **hash-table-hash-function**  
  
```scheme```
  hash-table
  ```
  Answer the equivalence and hash function of hash-table, respectively.
```

**hash**  

```
hash obj [size]
```  

**string-hash**  

```
string-hash obj [size]
```  

**string-ci-hash**  

```
string-ci-hash obj [size]
```  

**hash-by-identity**  

```
hash-by-identity obj [size]
```  

Answer a hash value appropriate for equality predicate `equal?`, `string=?`, `string-ci=?`, and `eq?`, respectively.

**hash** is a backwards-compatible replacement for Guile’s built-in `hash`.

### 7.5.40 SRFI-71 - Extended let-syntax for multiple values

This SRFI shadows the forms for `let`, `let*`, and `letrec` so that they may accept multiple values. For example:

```
(use-modules (srfi srfi-71))

(let* ((x y (values 1 2))
       (z (+ x y)))
  (* z 2))
⇒ 6
```


### 7.5.41 SRFI-87 => in case clauses

Starting from version 2.0.6, Guile’s core case syntax supports `=>` in clauses, as specified by SRFI-87/R7RS. See Section 6.13.2 [Conditionals], page 299.

### 7.5.42 SRFI-88 Keyword Objects

SRFI-88 ([http://srfi.schemers.org/srfi-88/srfi-88.html](http://srfi.schemers.org/srfi-88/srfi-88.html)) provides **keyword objects**, which are equivalent to Guile’s keywords (see Section 6.6.7 [Keywords], page 174). SRFI-88 keywords can be entered using the *postfix keyword syntax*, which consists of an identifier
followed by : (see Section 6.18.2 [Scheme Read], page 382). SRFI-88 can be made available with:

(use-modules (srfi srfi-88))

Doing so installs the right reader option for keyword syntax, using (read-set! keywords 'postfix). It also provides the procedures described below.

keyword? obj  
[Scheme Procedure]  
Return #t if obj is a keyword. This is the same procedure as the same-named built-in procedure (see Section 6.6.7.4 [Keyword Procedures], page 176).

(keyword? foo:) ⇒ #t
(keyword? 'foo:) ⇒ #t
(keyword? "foo") ⇒ #f

keyword->string kw  
[Scheme Procedure]  
Return the name of kw as a string, i.e., without the trailing colon. The returned string may not be modified, e.g., with string-set!.

(keyword->string foo:) ⇒ "foo"

string->keyword str  
[Scheme Procedure]  
Return the keyword object whose name is str.

(keyword->string (string->keyword "a b c")) ⇒ "a b c"

7.5.43 SRFI-98 Accessing environment variables.

This is a portable wrapper around Guile’s built-in support for interacting with the current environment, See Section 7.2.6 [Runtime Environment], page 512.

get-environment-variable name  
[Scheme Procedure]  
Returns a string containing the value of the environment variable given by the string name, or #f if the named environment variable is not found. This is equivalent to (getenv name).

get-environment-variables  
[Scheme Procedure]  
Returns the names and values of all the environment variables as an association list in which both the keys and the values are strings.

7.5.44 SRFI-105 Curly-infix expressions.

Guile’s built-in reader includes support for SRFI-105 curly-infix expressions. See the specification of SRFI-105 (http://srfi.schemers.org/srfi-105/srfi-105.html). Some examples:

{n <= 5} ⇒ (<= n 5)
{a + b + c} ⇒ (+ a b c)
{a * {b + c}} ⇒ (* a (+ b c))
{(- a) / b} ⇒ (/ (- a) b)
{-(a) / b} ⇒ (/ (- a) b) as well
{(f a b) + (g b)} ⇒ (+ (f a b) (g b)) as well
{f[a b] + g(h)} ⇒ (+ ($bracket-apply$ f a b) (g h))
To enable curly-infix expressions within a file, place the reader directive `#!curly-infix` before the first use of curly-infix notation. To globally enable curly-infix expressions in Guile’s reader, set the `curly-infix` read option.

Guile also implements the following non-standard extension to SRFI-105: if `curly-infix` is enabled and there is no other meaning assigned to square brackets (i.e. the `square-brackets` read option is turned off), then lists within square brackets are read as normal lists but with the special symbol `$bracket-list$` added to the front. To enable this combination of read options within a file, use the reader directive `#!curly-infix-and-bracket-lists`. For example:

\[
\begin{align*}
[a\ b] & \Rightarrow ($bracket-list$ a\ b) \\
[a\ .\ b] & \Rightarrow ($bracket-list$ a\ .\ b)
\end{align*}
\]

For more information on reader options, See Section 6.18.2 [Scheme Read], page 382.

### 7.5.45 SRFI-111 Boxes.


- **box** value
  
  Return a newly allocated box whose contents is initialized to `value`.

- **box?** obj
  
  Return true if `obj` is a box, otherwise return false.

- **unbox** box
  
  Return the current contents of `box`.

- **set-box!** box value
  
  Set the contents of `box` to `value`.

### 7.6 R6RS Support

See Section 6.20.6 [R6RS Libraries], page 414, for more information on how to define R6RS libraries, and their integration with Guile modules.

#### 7.6.1 Incompatibilities with the R6RS

There are some incompatibilities between Guile and the R6RS. Some of them are intentional, some of them are bugs, and some are simply unimplemented features. Please let the Guile developers know if you find one that is not on this list.
• The R6RS specifies many situations in which a conforming implementation must signal a specific error. Guile doesn’t really care about that too much—if a correct R6RS program would not hit that error, we don’t bother checking for it.
• Multiple library forms in one file are not yet supported. This is because the expansion of library sets the current module, but does not restore it. This is a bug.
• R6RS unicode escapes within strings are disabled by default, because they conflict with Guile’s already-existing escapes. The same is the case for R6RS treatment of escaped newlines in strings.

R6RS behavior can be turned on via a reader option. See Section 6.6.5.1 [String Syntax], page 141, for more information.
• A set! to a variable transformer may only expand to an expression, not a definition—even if the original set! expression was in definition context.
• Instead of using the algorithm detailed in chapter 10 of the R6RS, expansion of toplevel forms happens sequentially.

For example, while the expansion of the following set of toplevel definitions does the correct thing:

```
(begin
  (define even?
    (lambda (x)
      (or (= x 0) (odd? (- x 1)))))
  (define-syntax odd?
    (syntax-rules ()
      ((odd? x) (not (even? x)))))
  (even? 10))
⇒ #t
```

The same definitions outside of the begin wrapper do not:

```
(define even?
  (lambda (x)
    (or (= x 0) (odd? (- x 1)))))
(define-syntax odd?
  (syntax-rules ()
    ((odd? x) (not (even? x)))))
(even? 10)
<unnamed port>:4:18: In procedure even?:
<unnamed port>:4:18: Wrong type to apply: #<syntax-transformer odd?>
```

This is because when expanding the right-hand-side of even?, the reference to odd? is not yet marked as a syntax transformer, so it is assumed to be a function.

This bug will only affect top-level programs, not code in library forms. Fixing it for toplevel forms seems doable, but tricky to implement in a backward-compatible way. Suggestions and/or patches would be appreciated.

• The (rnrs io ports) module is incomplete. Work is ongoing to fix this.
• Guile does not prevent use of textual I/O procedures on binary ports, or vice versa. All ports in Guile support both binary and textual I/O. See Section 6.14.3 [Encoding], page 331, for full details.
• Guile’s implementation of \texttt{equal?} may fail to terminate when applied to arguments containing cycles.

7.6.2 R6RS Standard Libraries

In contrast with earlier versions of the Revised Report, the R6RS organizes the procedures and syntactic forms required of conforming implementations into a set of “standard libraries” which can be imported as necessary by user programs and libraries. Here we briefly list the libraries that have been implemented for Guile.

We do not attempt to document these libraries fully here, as most of their functionality is already available in Guile itself. The expectation is that most Guile users will use the well-known and well-documented Guile modules. These R6RS libraries are mostly useful to users who want to port their code to other R6RS systems.

The documentation in the following sections reproduces some of the content of the library section of the Report, but is mostly intended to provide supplementary information about Guile’s implementation of the R6RS standard libraries. For complete documentation, design rationales and further examples, we advise you to consult the “Standard Libraries” section of the Report (see Section “Standard Libraries” in \textit{The Revised^6 Report on the Algorithmic Language Scheme}).

7.6.2.1 Library Usage

Guile implements the R6RS ‘library’ form as a transformation to a native Guile module definition. As a consequence of this, all of the libraries described in the following subsections, in addition to being available for use by R6RS libraries and top-level programs, can also be imported as if they were normal Guile modules—via a \texttt{use-modules} form, say. For example, the R6RS “composite” library can be imported by:

\begin{verbatim}
(import (rnrs (6)))
(use-modules ((rnrs) :version (6)))
\end{verbatim}

For more information on Guile’s library implementation, see (see Section 6.20.6 [R6RS Libraries], page 414).

7.6.2.2 rnrs base

The \texttt{(rnrs base (6))} library exports the procedures and syntactic forms described in the main section of the Report (see Section “Base library” in \textit{The Revised^6 Report on the Algorithmic Language Scheme}). They are grouped below by the existing manual sections to which they correspond.

\begin{verbatim}
boolean? obj
not x
\end{verbatim}

See Section 6.6.1 [Booleans], page 104, for documentation.

\begin{verbatim}
symbol? obj
symbol->string sym
string->symbol str
\end{verbatim}

See Section 6.6.6.4 [Symbol Primitives], page 167, for documentation.

\begin{verbatim}
char? obj
char=?
\end{verbatim}

See Section 6.6.4 [Characters], page 157, for documentation.
char<?  
char>?  
char<=?  
char=>?  
integer->char n  
char->integer chr

See Section 6.6.3 [Characters], page 129, for documentation.

list? x  
null? x

See Section 6.6.9.2 [List Predicates], page 181, for documentation.

pair? x  
cons x y  
car pair  
cdr pair  
caar pair  
cadr pair  
cdar pair  
cddr pair  
caaar pair  
caadr pair  
caaddr pair  
caaaar pair  
caaaaaar pair  

See Section 6.6.8 [Pairs], page 178, for documentation.

number? obj  

See Section 6.6.2.1 [Numerical Tower], page 105, for documentation.
string? obj

See Section 6.6.5.2 [String Predicates], page 143, for documentation.

procedure? obj

See Section 6.9.7 [Procedure Properties], page 258, for documentation.

define name value

See Section 3.1.3 [Definition], page 16, for documentation.

set! variable-name value

See Section 6.9.7 [Procedure Properties], page 258, for documentation.

define-syntax keyword expression

See Section 6.10.1 [Defining Macros], page 261, for documentation.

let-syntax ((keyword transformer) . . .) exp1 exp2 . . .

See Section 6.10.1 [Defining Macros], page 261, for documentation.

letrec-syntax ((keyword transformer) . . .) exp1 exp2 . . .

See Section 6.10.1 [Defining Macros], page 261, for documentation.

let-values bindings body

See Section 6.12.2 [Local Bindings], page 295, for documentation.

let*-values bindings body

See Section 6.12.2 [Local Bindings], page 295, for documentation.

begin expr1 expr2 ...

See Section 6.13.1 [begin], page 298, for documentation.

quote expr

See Section 6.13.1 [begin], page 298, for documentation.

quasiquote expr

See Section 6.13.1 [begin], page 298, for documentation.

unquote expr

See Section 6.18.1.1 [Expression Syntax], page 379, for documentation.

unquote-splicing expr

See Section 6.18.1.1 [Expression Syntax], page 379, for documentation.

if test consequence [alternate]

See Section 6.13.2 [Conditionals], page 299, for documentation.

cond clause1 clause2 ...

See Section 6.13.2 [Conditionals], page 299, for documentation.

case key clause1 clause2 ...

See Section 6.13.2 [Conditionals], page 299, for documentation.

and expr ...

See Section 6.13.3 [and or], page 301, for documentation.

or expr ...

See Section 6.13.3 [and or], page 301, for documentation.
Chapter 7: Guile Modules

éq? x y  [Scheme Procedure]
eqv? x y  [Scheme Procedure]
equal? x y  [Scheme Procedure]
symbol=? symbol1 symbol2 ...  
See Section 6.11.1 [Equality], page 283, for documentation.

symbol=? is identical to eq?.

complex? z  [Scheme Procedure]
See Section 6.6.2.4 [Complex Numbers], page 113, for documentation.

real-part z  [Scheme Procedure]
imag-part z  [Scheme Procedure]
make-rectangular real_part imaginary_part  [Scheme Procedure]
make-polar x y  [Scheme Procedure]
magnitude z  [Scheme Procedure]
angle z  [Scheme Procedure]
See Section 6.6.2.10 [Complex], page 118, for documentation.

sqrt z  [Scheme Procedure]
exp z  [Scheme Procedure]
expt z1 z2  [Scheme Procedure]
log z  [Scheme Procedure]
sin z  [Scheme Procedure]
cos z  [Scheme Procedure]
tan z  [Scheme Procedure]
asin z  [Scheme Procedure]
acos z  [Scheme Procedure]
atan z  [Scheme Procedure]
See Section 6.6.2.12 [Scientific], page 123, for documentation.

real? x  [Scheme Procedure]
rational? x  [Scheme Procedure]
numerator x  [Scheme Procedure]
denominator x  [Scheme Procedure]
rationalize x eps  [Scheme Procedure]
See Section 6.6.2.3 [Reals and Rationals], page 110, for documentation.

exact? x  [Scheme Procedure]
inexact? x  [Scheme Procedure]
exact z  [Scheme Procedure]
inexact z  [Scheme Procedure]
See Section 6.6.2.5 [Exactness], page 113, for documentation. The exact and inexact procedures are identical to the inexact->exact and exact->inexact procedures provided by Guile’s code library.

integer? x  [Scheme Procedure]
See Section 6.6.2.2 [Integers], page 106, for documentation.
odd? $n$  

even? $n$  

gcd $x$ ...  

lcm $x$ ...  

exact-integer-sqrt $k$

See Section 6.6.2.7 [Integer Operations], page 116, for documentation.

$=$  

$<$  

$>$  

$<=$  

$>=$  

zero? $x$  

positive? $x$  

negative? $x$

See Section 6.6.2.8 [Comparison], page 117, for documentation.

for-each $f$ lst1 lst2 ...

See Section 7.5.3.5 [SRFI-1 Fold and Map], page 582, for documentation.

list $elem$ ...

See Section 6.6.9.3 [List Constructors], page 182, for documentation.

length lst  

list-ref lst $k$  

list-tail lst $k$

See Section 6.6.9.4 [List Selection], page 183, for documentation.

append lst ... obj

append

reverse lst

See Section 6.6.9.5 [Append/Reverse], page 183, for documentation.

number->string $n$ [radix]  

string->number str [radix]

See Section 6.6.2.9 [Conversion], page 118, for documentation.

string $char$ ...  

make-string $k$ [chr]

list->string lst

See Section 6.6.5.3 [String Constructors], page 144, for documentation.

string->list str [start [end]]

See Section 6.6.5.4 [List/String Conversion], page 145, for documentation.

string-length str  

string-ref str $k$  

string-copy str [start [end]]  

substring str start [end]

See Section 6.6.5.5 [String Selection], page 145, for documentation.
string=? s1 s2 s3 ...  [Scheme Procedure]
string<? s1 s2 s3 ...  [Scheme Procedure]
string>? s1 s2 s3 ...  [Scheme Procedure]
string<=? s1 s2 s3 ...  [Scheme Procedure]
string=>? s1 s2 s3 ...  [Scheme Procedure]

See Section 6.6.5.7 [String Comparison], page 148, for documentation.

string-append arg ...  [Scheme Procedure]

See Section 6.6.5.10 [Reversing and Appending Strings], page 155, for documentation.

string-for-each proc s [start [end]]  [Scheme Procedure]

See Section 6.6.5.11 [Mapping Folding and Unfolding], page 156, for documentation.

+ z1 ...
- z1 z2 ...
* z1 ...
/ z1 z2 ...
max x1 x2 ...
min x1 x2 ...
abs x
truncate x
floor x
ceiling x
round x

See Section 6.6.2.11 [Arithmetic], page 119, for documentation.

div x y  [Scheme Procedure]
mod x y  [Scheme Procedure]
div-and-mod x y  [Scheme Procedure]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. 
div returns the integer q and mod returns the real number r such that
x = q * y + r and 0 <= r < abs(y). 
div-and-mod returns both q and r, and is more efficient than computing each separately. 
Note that when y > 0, div returns floor(x/y), otherwise it returns ceiling(x/y).

(div 123 10) ⇒ 12
(mod 123 10) ⇒ 3
(div-and-mod 123 10) ⇒ 12 and 3
(div-and-mod 123 -10) ⇒ -12 and 3
(div-and-mod -123 10) ⇒ -13 and 7
(div-and-mod -123 -10) ⇒ 13 and 7
(div-and-mod -123.2 -63.5) ⇒ 2.0 and 3.8
(div-and-mod 16/3 -10/7) ⇒ -3 and 22/21

div0 x y  [Scheme Procedure]
mod0 x y  [Scheme Procedure]
div0-and-mod0 x y  [Scheme Procedure]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. 
div0 returns the integer q and mod0 returns the real number r such that

See Section 6.6.2.11 [Arithmetic], page 119, for documentation.
\[ x = q \cdot y + r \text{ and } -\text{abs}(y/2) \leq r < \text{abs}(y/2). \]
\text{div0-and-mod0} returns both \( q \) and \( r \), and is more efficient than computing each separately.

Note that \text{div0} returns \( x/y \) rounded to the nearest integer. When \( x/y \) lies exactly half-way between two integers, the tie is broken according to the sign of \( y \). If \( y > 0 \), ties are rounded toward positive infinity, otherwise they are rounded toward negative infinity. This is a consequence of the requirement that \(-\text{abs}(y/2) \leq r < \text{abs}(y/2)\).

\[
\begin{align*}
(\text{div0} 123 10) &\Rightarrow 12 \\
(\text{mod0} 123 10) &\Rightarrow 3 \\
(\text{div0-and-mod0} 123 10) &\Rightarrow 12 \text{ and } 3 \\
(\text{div0-and-mod0} 123 -10) &\Rightarrow -12 \text{ and } 3 \\
(\text{div0-and-mod0} -123 10) &\Rightarrow -12 \text{ and } -3 \\
(\text{div0-and-mod0} -123 -10) &\Rightarrow 12 \text{ and } -3 \\
(\text{div0-and-mod0} -123.2 -63.5) &\Rightarrow 2.0 \text{ and } 3.8 \\
(\text{div0-and-mod0} 16/3 -10/7) &\Rightarrow -4 \text{ and } -8/21
\end{align*}
\]

\textbf{real-valued? \ obj} \quad \textbf{[Scheme Procedure]}\n
\textbf{rational-valued? \ obj} \quad \textbf{[Scheme Procedure]}\n
\textbf{integer-valued? \ obj} \quad \textbf{[Scheme Procedure]}\n
These procedures return \#t if and only if their arguments can, respectively, be coerced to a real, rational, or integer value without a loss of numerical precision.

\textbf{real-valued?} will return \#t for complex numbers whose imaginary parts are zero.

\textbf{nan? \ x} \quad \textbf{[Scheme Procedure]}\n
\textbf{infinite? \ x} \quad \textbf{[Scheme Procedure]}\n
\textbf{finite? \ x} \quad \textbf{[Scheme Procedure]}\n
\texttt{nan?} returns \#t if \( x \) is a NaN value, \#f otherwise. \texttt{infinite?} returns \#t if \( x \) is an infinite value, \#f otherwise. \texttt{finite?} returns \#t if \( x \) is neither infinite nor a NaN value, otherwise it returns \#f. Every real number satisfies exactly one of these predicates. An exception is raised if \( x \) is not real.

\textbf{assert \ expr} \quad \textbf{[Scheme Syntax]}\n
Raises an \&assertion condition if \( expr \) evaluates to \#f; otherwise evaluates to the value of \( expr \).

\textbf{error \ who \ message \ irritant1 \ ...} \quad \textbf{[Scheme Procedure]}\n
\textbf{assertion-violation \ who \ message \ irritant1 \ ...} \quad \textbf{[Scheme Procedure]}\n
These procedures raise compound conditions based on their arguments: If \( who \) is not \#f, the condition will include a \&who condition whose \who field is set to \( who \); a \&message condition will be included with a \message field equal to \( message \); an \&irritants condition will be included with its \irritants list given by \( irritant1 \) ... .

\texttt{error} produces a compound condition with the simple conditions described above, as well as an \&error condition; \texttt{assertion-violation} produces one that includes an \&assertion condition.

\textbf{vector-map \ proc \ v} \quad \textbf{[Scheme Procedure]}\n
\textbf{vector-for-each \ proc \ v} \quad \textbf{[Scheme Procedure]}\n
These procedures implement the \texttt{map} and \texttt{for-each} contracts over vectors.
### 7.6.2.3 rnrs unicode

The (rnrs unicode (6)) library provides procedures for manipulating Unicode characters and strings.

<table>
<thead>
<tr>
<th>Scheme Procedure</th>
<th>Scheme Procedure</th>
<th>Scheme Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>char-upcase char</td>
<td>char-downcase char</td>
<td>char-titlecase char</td>
</tr>
<tr>
<td>char-foldcase char</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These procedures translate their arguments from one Unicode character set to another. **char-upcase**, **char-downcase**, and **char-titlecase** are identical to their counterparts in the Guile core library; See Section 6.6.3 [Characters], page 129, for documentation.

**char-foldcase** returns the result of applying **char-upcase** to its argument, followed by **char-downcase**—except in the case of the Turkic characters U+0130 and U+0131, for which the procedure acts as the identity function.

<table>
<thead>
<tr>
<th>Scheme Procedure</th>
<th>Scheme Procedure</th>
<th>Scheme Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>char-ci=? char1 char2 char3 ...</td>
<td>char-ci&lt;? char1 char2 char3 ...</td>
<td>char-ci&gt;? char1 char2 char3 ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char-ci&lt;=? char1 char2 char3 ...</td>
</tr>
</tbody>
</table>
These procedures facilitate case-insensitive comparison of Unicode characters. They
are identical to the procedures provided by Guile’s core library. See Section 6.6.3
[Characters], page 129, for documentation.

These procedures implement various Unicode character set predicates. They are iden-
tical to the procedures provided by Guile’s core library. See Section 6.6.3 [Characters],
page 129, for documentation.

These procedures perform Unicode case folding operations on their input. See
Section 6.6.5.9 [Alphabetic Case Mapping], page 154, for documentation.

These procedures perform case-insensitive comparison on their input. See
Section 6.6.5.7 [String Comparison], page 148, for documentation.

These procedures perform Unicode string normalization operations on their input.
See Section 6.6.5.7 [String Comparison], page 148, for documentation.

The (rnrs bytevectors (6)) library provides procedures for working with blocks of binary
data. This functionality is documented in its own section of the manual; See Section 6.6.12
[Bytevectors], page 193.

The (rnrs lists (6)) library provides procedures additional procedures for working with
lists.
**find proc list**  
This procedure is identical to the one defined in Guile’s SRFI-1 implementation. See Section 7.5.3.7 [SRFI-1 Searching], page 586, for documentation.

**for-all proc list1 list2 ...**  
**exists proc list1 list2 ...**  
The for-all procedure is identical to the every procedure defined by SRFI-1; the exists procedure is identical to SRFI-1’s any. See Section 7.5.3.7 [SRFI-1 Searching], page 586, for documentation.

**filter proc list**  
**partition proc list**  
These procedures are identical to the ones provided by SRFI-1. See Section 6.6.9.6 [List Modification], page 184, for a description of filter; See Section 7.5.3.6 [SRFI-1 Filtering and Partitioning], page 585, for partition.

**fold-right combine nil list1 list2 . . .**  
**fold-left combine nil list1 list2 . . .**  
This procedure is identical the fold-right procedure provided by SRFI-1. See Section 7.5.3.5 [SRFI-1 Fold and Map], page 582, for documentation.

**remp proc list**  
**remove obj list**  
**remv obj list**  
**remq obj list**  
remove, remv, and remq are identical to the delete, delv, and delq procedures provided by Guile’s core library, (see Section 6.6.9.6 [List Modification], page 184). remp is identical to the alternate remove procedure provided by SRFI-1; See Section 7.5.3.8 [SRFI-1 Deleting], page 587.

**memp proc list**  
**member obj list**  
**memv obj list**  
**memq obj list**  
member, memv, and memq are identical to the procedures provided by Guile’s core library; See Section 6.6.9.7 [List Searching], page 185, for their documentation. memp uses the specified predicate function proc to test elements of the list list—it behaves similarly to find, except that it returns the first sublist of list whose car satisfies proc.

**assp proc alist**  
**assoc obj alist**  
**assv obj alist**  
[Scheme Procedure]
assq obj alist

assoc, assv, and assq are identical to the procedures provided by Guile’s core library; See Section 6.6.20.1 [Alist Key Equality], page 230, for their documentation. assq uses the specified predicate function proc to test keys in the association list alist.

cons* obj1 ... obj

This procedure is identical to the one exported by Guile’s core library. See Section 6.6.9.3 [List Constructors], page 182, for documentation.

7.6.2.6 rnrs sorting

The (rnrs sorting (6)) library provides procedures for sorting lists and vectors.

list-sort proc list
vector-sort proc vector

These procedures return their input sorted in ascending order, without modifying the original data. proc must be a procedure that takes two elements from the input list or vector as arguments, and returns a true value if the first is “less” than the second, #f otherwise. list-sort returns a list; vector-sort returns a vector.

Both list-sort and vector-sort are implemented in terms of the stable-sort procedure from Guile’s core library. See Section 6.11.3 [Sorting], page 286, for a discussion of the behavior of that procedure.

vector-sort! proc vector

Performs a destructive, “in-place” sort of vector, using proc as described above to determine an ascending ordering of elements. vector-sort! returns an unspecified value.

This procedure is implemented in terms of the sort! procedure from Guile’s core library. See Section 6.11.3 [Sorting], page 286, for more information.

7.6.2.7 rnrs control

The (rnrs control (6)) library provides syntactic forms useful for constructing conditional expressions and controlling the flow of execution.

when test expression1 expression2 ...
unless test expression1 expression2 ...

do ((variable init step) ...) (test expression ...) command ...

case-lambda clause ...

This form is identical to the one provided by Guile’s core library. See Section 6.9.5 [Case-lambda], page 255, for documentation.
7.6.2.8 R6RS Records

The manual sections below describe Guile’s implementation of R6RS records, which provide support for user-defined data types. The R6RS records API provides a superset of the features provided by Guile’s “native” records, as well as those of the SRFI-9 records API; See Section 6.6.17 [Records], page 221, and Section 6.6.16 [SRFI-9 Records], page 218, for a description of those interfaces.

As with SRFI-9 and Guile’s native records, R6RS records are constructed using a record-type descriptor that specifies attributes like the record’s name, its fields, and the mutability of those fields.

R6RS records extend this framework to support single inheritance via the specification of a “parent” type for a record type at definition time. Accessors and mutator procedures for the fields of a parent type may be applied to records of a subtype of this parent. A record type may be sealed, in which case it cannot be used as the parent of another record type.

The inheritance mechanism for record types also informs the process of initializing the fields of a record and its parents. Constructor procedures that generate new instances of a record type are obtained from a record constructor descriptor, which encapsulates the record-type descriptor of the record to be constructed along with a protocol procedure that defines how constructors for record subtypes delegate to the constructors of their parent types.

A protocol is a procedure used by the record system at construction time to bind arguments to the fields of the record being constructed. The protocol procedure is passed a procedure n that accepts the arguments required to construct the record’s parent type; this procedure, when invoked, will return a procedure p that accepts the arguments required to construct a new instance of the record type itself and returns a new instance of the record type.

The protocol should in turn return a procedure that uses n and p to initialize the fields of the record type and its parent type(s). This procedure will be the constructor returned by

As a trivial example, consider the hypothetical record type `pixel`, which encapsulates an x-y location on a screen, and `voxel`, which has `pixel` as its parent type and stores an additional coordinate. The following protocol produces a constructor procedure that accepts all three coordinates, uses the first two to initialize the fields of `pixel`, and binds the third to the single field of `voxel`.

```
(lambda (n)
  (lambda (x y z)
    (let ((p (n x y)))
      (p z)))
```

It may be helpful to think of protocols as “constructor factories” that produce chains of delegating constructors glued together by the helper procedure n.

An R6RS record type may be declared to be nongenerative via the use of a unique generated or user-supplied symbol—or uid—such that subsequent record type declarations with the same uid and attributes will return the previously-declared record-type descriptor.
R6RS record types may also be declared to be opaque, in which case the various predicates and introspection procedures defined in (rnrs records introspection) will behave as if records of this type are not records at all.

Note that while the R6RS records API shares much of its namespace with both the SRFI-9 and native Guile records APIs, it is not currently compatible with either.

### 7.6.2.9 rnrs records syntactic

The (rnrs records syntactic (6)) library exports the syntactic API for working with R6RS records.

```
define-record-type name-spec record-clause ... [Scheme Syntax]
```

Defines a new record type, introducing bindings for a record-type descriptor, a record constructor descriptor, a constructor procedure, a record predicate, and accessor and mutator procedures for the new record type’s fields.

- **name-spec** must either be an identifier or must take the form (record-name constructor-name predicate-name), where record-name, constructor-name, and predicate-name are all identifiers and specify the names to which, respectively, the record-type descriptor, constructor, and predicate procedures will be bound.
- If name-spec is only an identifier, it specifies the name to which the generated record-type descriptor will be bound.

Each **record-clause** must be one of the following:

- (fields field-spec*), where each field-spec specifies a field of the new record type and takes one of the following forms:
  - (immutable field-name accessor-name), which specifies an immutable field with the name field-name and binds an accessor procedure for it to the name given by accessor-name
  - (mutable field-name accessor-name mutator-name), which specifies a mutable field with the name field-name and binds accessor and mutator procedures to accessor-name and mutator-name, respectively
  - (immutable field-name), which specifies an immutable field with the name field-name; an accessor procedure for it will be created and named by appending record name and field-name with a hyphen separator
  - (mutable field-name), which specifies a mutable field with the name field-name; an accessor procedure for it will be created and named as described above; a mutator procedure will also be created and named by appending -set! to the accessor name
  - field-name, which specifies an immutable field with the name field-name; an access procedure for it will be created and named as described above
- (parent parent-name), where parent-name is a symbol giving the name of the record type to be used as the parent of the new record type
- (protocol expression), where expression evaluates to a protocol procedure which behaves as described above, and is used to create a record constructor descriptor for the new record type
- (sealed sealed?), where sealed? is a boolean value that specifies whether or not the new record type is sealed
• (opaque opaque?), where opaque? is a boolean value that specifies whether or not the new record type is opaque
• (nongenerative [uid]), which specifies that the record type is nongenerative via the optional uid uid. If uid is not specified, a unique uid will be generated at expansion time
• (parent-rtd parent-rtd parent-cd), a more explicit form of the parent form above; parent-rtd and parent-cd should evaluate to a record-type descriptor and a record constructor descriptor, respectively

record-type-descriptor record-name [Scheme Syntax]
Evaluates to the record-type descriptor associated with the type specified by record-name.

record-constructor-descriptor record-name [Scheme Syntax]
Evaluates to the record-constructor descriptor associated with the type specified by record-name.

7.6.2.10 rnrs records procedural
The (rnrs records procedural (6)) library exports the procedural API for working with R6RS records.

make-record-type-descriptor name parent uid sealed? opaque? fields
Returns a new record-type descriptor with the specified characteristics: name must be a symbol giving the name of the new record type; parent must be either #f or a non-sealed record-type descriptor for the returned record type to extend; uid must be either #f, indicating that the record type is generative, or a symbol giving the type’s nongenerative uid; sealed? and opaque? must be boolean values that specify the sealedness and opaqueness of the record type; fields must be a vector of zero or more field specifiers of the form (mutable name) or (immutable name), where name is a symbol giving a name for the field.
If uid is not #f, it must be a symbol

record-type-descriptor? obj [Scheme Procedure]
Returns #t if obj is a record-type descriptor, #f otherwise.

make-record-constructor-descriptor rtd parent-constructor-descriptor protocol
Returns a new record constructor descriptor that can be used to produce constructors for the record type specified by the record-type descriptor rtd and whose delegation and binding behavior are specified by the protocol procedure protocol.
parent-constructor-descriptor specifies a record constructor descriptor for the parent type of rtd, if one exists. If rtd represents a base type, then parent-constructor-descriptor must be #f. If rtd is an extension of another type, parent-constructor-descriptor may still be #f, but protocol must also be #f in this case.

record-constructor rcd [Scheme Procedure]
Returns a record constructor procedure by invoking the protocol defined by the record-constructor descriptor rcd.
**record-predicate rtd**  
Returns the record predicate procedure for the record-type descriptor `rtd`.

**record-accessor rtd k**  
Returns the record field accessor procedure for the `k`th field of the record-type descriptor `rtd`.

**record-mutator rtd k**  
Returns the record field mutator procedure for the `k`th field of the record-type descriptor `rtd`. An `&assertion` condition will be raised if this field is not mutable.

### 7.6.2.11 rnrs records inspection

The `rnrs records inspection` library provides procedures useful for accessing metadata about R6RS records.

**record? obj**  
Return `#t` if the specified object is a non-opaque R6RS record, `#f` otherwise.

**record-rtd record**  
Returns the record-type descriptor for `record`. An `&assertion` is raised if `record` is opaque.

**record-type-name rtd**  
Returns the name of the record-type descriptor `rtd`.

**record-type-parent rtd**  
Returns the parent of the record-type descriptor `rtd`, or `#f` if it has none.

**record-type-uid rtd**  
Returns the uid of the record-type descriptor `rtd`, or `#f` if it has none.

**record-type-generative? rtd**  
Returns `#t` if the record-type descriptor `rtd` is generative, `#f` otherwise.

**record-type-sealed? rtd**  
Returns `#t` if the record-type descriptor `rtd` is sealed, `#f` otherwise.

**record-type-opaque? rtd**  
Returns `#t` if the record-type descriptor `rtd` is opaque, `#f` otherwise.

**record-type-field-names rtd**  
Returns a vector of symbols giving the names of the fields defined by the record-type descriptor `rtd` (and not any of its sub- or supertypes).

**record-field-mutable? rtd k**  
Returns `#t` if the field at index `k` of the record-type descriptor `rtd` (and not any of its sub- or supertypes) is mutable.
7.6.2.12 rnrs exceptions

The (rnrs exceptions (6)) library provides functionality related to signaling and handling exceptional situations. This functionality is similar to the exception handling systems provided by Guile's core library See Section 6.13.8 [Exceptions], page 311, and by the SRFI-18 and SRFI-34 modules—See Section 7.5.15.5 [SRFI-18 Exceptions], page 607, and Section 7.5.23 [SRFI-34], page 619, respectively—but there are some key differences in concepts and behavior.

A raised exception may be **continuable** or **non-continuable**. When an exception is raised non-continuously, another exception, with the condition type &non-continuable, will be raised when the exception handler returns locally. Raising an exception continuously captures the current continuation and invokes it after a local return from the exception handler.

Like SRFI-18 and SRFI-34, R6RS exceptions are implemented on top of Guile's native throw and catch forms, and use custom “throw keys” to identify their exception types. As a consequence, Guile’s catch form can handle exceptions thrown by these APIs, but the reverse is not true: Handlers registered by the with-exception-handler procedure described below will only be called on exceptions thrown by the corresponding raise procedure.

### with-exception-handler handler thunk

Installs handler, which must be a procedure taking one argument, as the current exception handler during the invocation of thunk, a procedure taking zero arguments. The handler in place at the time with-exception-handler is called is made current again once either thunk returns or handler is invoked after an exception is thrown from within thunk.

This procedure is similar to the with-throw-handler procedure provided by Guile’s code library; (see Section 6.13.8.3 [Throw Handlers], page 314).

### guard (variable clause1 clause2 ...) body

Evaluates the expression given by body, first creating an ad hoc exception handler that binds a raised exception to variable and then evaluates the specified clauses as if they were part of a cond expression, with the value of the first matching clause becoming the value of the guard expression (see Section 6.13.2 [Conditionals], page 299). If none of the clause’s test expressions evaluates to #t, the exception is re-raised, with the exception handler that was current before the evaluation of the guard form.

For example, the expression

\[
\text{(guard (ex ((eq? ex 'foo) 'bar) ((eq? ex 'bar) 'baz)) (raise 'bar))}
\]

evaluates to baz.

### raise obj

Raises a non-continuable exception by invoking the currently-installed exception handler on obj. If the handler returns, a &non-continuable exception will be raised in the dynamic context in which the handler was installed.

### raise-continuable obj

Raises a continuable exception by invoking currently-installed exception handler on obj.
7.6.2.13 rnrs conditions

The (rnrs condition (6)) library provides forms and procedures for constructing new condition types, as well as a library of pre-defined condition types that represent a variety of common exceptional situations. Conditions are records of a subtype of the &condition record type, which is neither sealed nor opaque. See Section 7.6.2.8 [R6RS Records], page 661.

Conditions may be manipulated singly, as simple conditions, or when composed with other conditions to form compound conditions. Compound conditions do not “nest”—constructing a new compound condition out of existing compound conditions will “flatten” them into their component simple conditions. For example, making a new condition out of a &message condition and a compound condition that contains an &assertion condition and another &message condition will produce a compound condition that contains two &message conditions and one &assertion condition.

The record type predicates and field accessors described below can operate on either simple or compound conditions. In the latter case, the predicate returns #t if the compound condition contains a component simple condition of the appropriate type; the field accessors return the requisite fields from the first component simple condition found to be of the appropriate type.

This library is quite similar to the SRFI-35 conditions module (see Section 7.5.24 [SRFI-35], page 620). Among other minor differences, the (rnrs conditions) library features slightly different semantics around condition field accessors, and comes with a larger number of pre-defined condition types. The two APIs are not currently compatible, however; the condition? predicate from one API will return #f when applied to a condition object created in the other.

&condition  [Condition Type]
condition? obj  [Scheme Procedure]
The base record type for conditions.

condition condition1 ...  [Scheme Procedure]
simple-conditions condition  [Scheme Procedure]
The condition procedure creates a new compound condition out of its condition arguments, flattening any specified compound conditions into their component simple conditions as described above.

simple-conditions returns a list of the component simple conditions of the compound condition condition, in the order in which they were specified at construction time.

condition-predicate rtd  [Scheme Procedure]
condition-accessor rtd proc  [Scheme Procedure]
These procedures return condition predicate and accessor procedures for the specified condition record type rtd.

define-condition-type condition-type supertype constructor  [Scheme Syntax]
predicate field-spec ...
Evaluates to a new record type definition for a condition type with the name condition-type that has the condition type supertype as its parent. A default constructor, which
binds its arguments to the fields of this type and its parent types, will be bound to
the identifier constructor; a condition predicate will be bound to predicate. The fields
of the new type, which are immutable, are specified by the field-specs, each of which
must be of the form:

\[(\text{field accessor})\]

where field gives the name of the field and accessor gives the name for a binding to
an accessor procedure created for this field.

\&message
\make-message-condition message
\message-condition? obj
\condition-message condition

A type that includes a message describing the condition that occurred.

\&warning
\make-warning
\warning? obj

A base type for representing non-fatal conditions during execution.

\&serious
\make-serious-condition
\serious-condition? obj

A base type for conditions representing errors serious enough that cannot be ignored.

\&error
\make-error
\error? obj

A base type for conditions representing errors.

\&violation
\make-violation
\violation?

A subtype of \&serious that can be used to represent violations of a language or
library standard.

\&assertion
\make-assertion-violation
\assertion-violation? obj

A subtype of \&violation that indicates an invalid call to a procedure.

\&irritants
\make-irritants-condition irritants
\irritants-condition? obj
\condition-irritants condition

A base type used for storing information about the causes of another condition in a
compound condition.

\&who
\make-who-condition who
who-condition? obj

A base type used for storing the identity, a string or symbol, of the entity responsible for another condition in a compound condition.

&non-continuable

A subtype of &violation used to indicate that an exception handler invoked by raise has returned locally.

&implementation-restriction

A subtype of &violation used to indicate a violation of an implementation restriction.

&lexical

A subtype of &violation used to indicate a syntax violation at the level of the datum syntax.

&syntax

A subtype of &violation that indicates a syntax violation. The form and subform fields, which must be datum values, indicate the syntactic form responsible for the condition.

&undefined

A subtype of &violation that indicates a reference to an unbound identifier.

7.6.2.14 I/O Conditions

These condition types are exported by both the (rnrs io ports (6)) and (rnrs io simple (6)) libraries.

&i/o

A condition supertype for more specific I/O errors.

&i/o-read

A subtype of &i/o; represents read-related I/O errors.
&i/o-write

make-i/o-write-error

i/o-write-error? obj

A subtype of &i/o; represents write-related I/O errors.

&i/o-invalid-position

make-i/o-invalid-position-error position

i/o-invalid-position-error? obj

i/o-error-position condition

A subtype of &i/o; represents an error related to an attempt to set the file position to an invalid position.

&i/o-filename

make-i/o-filename-error filename

i/o-filename-error? obj

i/o-error-filename condition

A subtype of &i/o; represents an error related to an operation on a named file.

&i/o-file-protection

make-i/o-file-protection-error filename

i/o-file-protection-error? obj

A subtype of &i/o-filename; represents an error resulting from an attempt to access a named file for which the caller had insufficient permissions.

&i/o-file-is-read-only

make-i/o-file-is-read-only-error filename

i/o-file-is-read-only-error? obj

A subtype of &i/o-file-protection; represents an error related to an attempt to write to a read-only file.

&i/o-file-already-exists

make-i/o-file-already-exists-error filename

i/o-file-already-exists-error? obj

A subtype of &i/o-filename; represents an error related to an operation on an existing file that was assumed not to exist.

&i/o-file-does-not-exist

make-i/o-file-does-not-exist-error

i/o-file-does-not-exist-error? obj

A subtype of &i/o-filename; represents an error related to an operation on a nonexistent file that was assumed to exist.

&i/o-port

make-i/o-port-error port

i/o-port-error? obj

i/o-error-port condition

A subtype of &i/o; represents an error related to an operation on the port port.
7.6.2.15 Transcoders

The transcoder facilities are exported by \texttt{(rnrs io ports)}.

Several different Unicode encoding schemes describe standard ways to encode characters and strings as byte sequences and to decode those sequences. Within this document, a codec is an immutable Scheme object that represents a Unicode or similar encoding scheme.

An end-of-line style is a symbol that, if it is not \texttt{none}, describes how a textual port transcodes representations of line endings.

A transcoder is an immutable Scheme object that combines a codec with an end-of-line style and a method for handling decoding errors. Each transcoder represents some specific bidirectional (but not necessarily lossless), possibly stateful translation between byte sequences and Unicode characters and strings. Every transcoder can operate in the input direction (bytes to characters) or in the output direction (characters to bytes). A transcoder parameter name means that the corresponding argument must be a transcoder.

A binary port is a port that supports binary I/O, does not have an associated transcoder and does not support textual I/O. A textual port is a port that supports textual I/O, and does not support binary I/O. A textual port may or may not have an associated transcoder.

\begin{verbatim}
latin-1-codec  [Scheme Procedure]
utf-8-codec    [Scheme Procedure]
utf-16-codec   [Scheme Procedure]
\end{verbatim}

These are predefined codecs for the ISO 8859-1, UTF-8, and UTF-16 encoding schemes.

A call to any of these procedures returns a value that is equal in the sense of \texttt{eqv?} to the result of any other call to the same procedure.

\begin{verbatim}
eol-style  [Scheme Syntax]
eol-style-symbol
\end{verbatim}

\texttt{eol-style-symbol} should be a symbol whose name is one of \texttt{lf}, \texttt{cr}, \texttt{crlf}, \texttt{nel}, \texttt{crnel}, \texttt{ls}, and \texttt{none}.

The form evaluates to the corresponding symbol. If the name of \texttt{eol-style-symbol} is not one of these symbols, the effect and result are implementation-dependent; in particular, the result may be an eol-style symbol acceptable as an \texttt{eol-style} argument to \texttt{make-transcoder}. Otherwise, an exception is raised.

All eol-style symbols except \texttt{none} describe a specific line-ending encoding:

\begin{verbatim}
lf     linefeed
cr     carriage return
crlf   carriage return, linefeed
nel    next line
crnel  carriage return, next line
ls     line separator
\end{verbatim}

For a textual port with a transcoder, and whose transcoder has an eol-style symbol \texttt{none}, no conversion occurs. For a textual input port, any eol-style symbol other than \texttt{none} means that all of the above line-ending encodings are recognized and are translated into a single linefeed. For a textual output port, \texttt{none} and \texttt{lf} are equivalent.
Linefeed characters are encoded according to the specified eol-style symbol, and all other characters that participate in possible line endings are encoded as is.

**Note:** Only the name of `eol-style-symbol` is significant.

---

**native-eol-style**

Returns the default end-of-line style of the underlying platform, e.g., `lf` on Unix and `crlf` on Windows.

---

**&i/o-decoding**

**make-i/o-decoding-error** `port`

**i/o-decoding-error?** `obj`

This condition type could be defined by

```
(define-condition-type &i/o-decoding &i/o-port
  make-i/o-decoding-error i/o-decoding-error?)
```

An exception with this type is raised when one of the operations for textual input from a port encounters a sequence of bytes that cannot be translated into a character or string by the input direction of the port’s transcoder.

When such an exception is raised, the port’s position is past the invalid encoding.

---

**&i/o-encoding**

**make-i/o-encoding-error** `port` `char`

**i/o-encoding-error?** `obj`

**i/o-encoding-error-char** `condition`

This condition type could be defined by

```
(define-condition-type &i/o-encoding &i/o-port
  make-i/o-encoding-error i/o-encoding-error?
  (char i/o-encoding-error-char))
```

An exception with this type is raised when one of the operations for textual output to a port encounters a character that cannot be translated into bytes by the output direction of the port’s transcoder. `char` is the character that could not be encoded.

---

**error-handling-mode**

**error-handling-mode-symbol**

`error-handling-mode-symbol` should be a symbol whose name is one of `ignore`, `raise`, and `replace`. The form evaluates to the corresponding symbol. If `error-handling-mode-symbol` is not one of these identifiers, effect and result are implementation-dependent: The result may be an error-handling-mode symbol acceptable as a `handling-mode` argument to `make-transcoder`. If it is not acceptable as a `handling-mode` argument to `make-transcoder`, an exception is raised.

**Note:** Only the name of `error-handling-mode-symbol` is significant.

The error-handling mode of a transcoder specifies the behavior of textual I/O operations in the presence of encoding or decoding errors.

If a textual input operation encounters an invalid or incomplete character encoding, and the error-handling mode is `ignore`, an appropriate number of bytes of the invalid encoding are ignored and decoding continues with the following bytes.

If the error-handling mode is `replace`, the replacement character `U+FFFD` is injected into the data stream, an appropriate number of bytes are ignored, and decoding continues with the following bytes.
If the error-handling mode is `raise`, an exception with condition type `&i/o-decoding` is raised.

If a textual output operation encounters a character it cannot encode, and the error-handling mode is `ignore`, the character is ignored and encoding continues with the next character. If the error-handling mode is `replace`, a codec-specific replacement character is emitted by the transcoder, and encoding continues with the next character. The replacement character is U+FFFD for transcoders whose codec is one of the Unicode encodings, but is the ? character for the Latin-1 encoding. If the error-handling mode is `raise`, an exception with condition type `&i/o-encoding` is raised.

\begin{verbatim}
make-transcoder codec              [Scheme Procedure]
make-transcoder codec eol-style    [Scheme Procedure]
make-transcoder codec eol-style handling-mode  [Scheme Procedure]
\end{verbatim}

codec must be a codec; eol-style, if present, an eol-style symbol; and handling-mode, if present, an error-handling-mode symbol.

eol-style may be omitted, in which case it defaults to the native end-of-line style of the underlying platform. handling-mode may be omitted, in which case it defaults to replace. The result is a transcoder with the behavior specified by its arguments.

\begin{verbatim}
native-transcoder                   [Scheme procedure]
\end{verbatim}

Returns an implementation-dependent transcoding that represents a possibly locale-dependent “native” transcoding.

\begin{verbatim}
transcoder-codec transcoder         [Scheme Procedure]
transcoder-eol-style transcoder    [Scheme Procedure]
transcoder-error-handling-mode transcoder [Scheme Procedure]
\end{verbatim}

These are accessors for transcoding objects; when applied to a transcoding returned by make-transcoder, they return the codec, eol-style, and handling-mode arguments, respectively.

\begin{verbatim}
bytevector->string bytevector transcoder         [Scheme Procedure]
\end{verbatim}

Returns the string that results from transcoding the bytevector according to the input direction of the transcoding.

\begin{verbatim}
string->bytevector string transcoder [Scheme Procedure]
\end{verbatim}

Returns the bytevector that results from transcoding the string according to the output direction of the transcoding.

### 7.6.2.16 rnrs io ports

Guile’s binary and textual port interface was heavily inspired by R6RS, so many R6RS port interfaces are documented elsewhere. Note that R6RS ports are not disjoint from Guile’s native ports, so Guile-specific procedures will work on ports created using the R6RS API, and vice versa. Also note that in Guile, all ports are both textual and binary. See Section 6.14 [Input and Output], page 328, for more on Guile’s core port API. The R6RS ports module wraps Guile’s I/O routines in a helper that will translate native Guile exceptions to R6RS conditions; See Section 7.6.2.14 [R6RS I/O Conditions], page 668, for more. See
Section 7.6.2.17 [R6RS File Ports], page 676, for documentation on the R6RS file port interface.

Note: The implementation of this R6RS API is not complete yet.

eof-object? obj

See Section 6.14.2 [Binary I/O], page 330, for documentation.

eof-object

Return the end-of-file (EOF) object.

(eof-object? (eof-object))
⇒ #t

port? obj
input-port? obj
output-port? obj

See Section 6.14.1 [Ports], page 328, for documentation.

port-transcorder port

Return a transcoder associated with the encoding of port. See Section 6.14.3 [Encoding], page 331, and See Section 7.6.2.15 [R6RS Transcoders], page 670.

binary-port? port

Return #t if port appears to be a binary port, else return #f. Note that Guile does not currently distinguish between binary and textual ports, so this predicate is not a reliable indicator of whether the port was created as a binary port. Currently, it returns #t if and only if the port encoding is “ISO-8859-1”, because Guile uses this encoding when creating a binary port. See Section 6.14.3 [Encoding], page 331, for more details.

textual-port? port

Return #t if port appears to be a textual port, else return #f. Note that Guile does not currently distinguish between binary and textual ports, so this predicate is not a reliable indicator of whether the port was created as a textual port. Currently, it always returns #t, because all ports can be used for textual I/O in Guile. See Section 6.14.3 [Encoding], page 331, for more details.

transcoded-port binary-port transcoder

The transcoded-port procedure returns a new textual port with the specified transcoder. Otherwise the new textual port’s state is largely the same as that of binary-port. If binary-port is an input port, the new textual port will be an input port and will transcode the bytes that have not yet been read from binary-port. If binary-port is an output port, the new textual port will be an output port and will transcode output characters into bytes that are written to the byte sink represented by binary-port.

As a side effect, however, transcoded-port closes binary-port in a special way that allows the new textual port to continue to use the byte source or sink represented by binary-port, even though binary-port itself is closed and cannot be used by the input and output operations described in this chapter.
port-position port
   Equivalent to (seek port SEEK_CUR 0). See Section 6.14.7 [Random Access], page 337.

port-has-port-position? port
   Return #t is port supports port-position.

set-port-position! port offset
   Equivalent to (seek port SEEK_SET offset). See Section 6.14.7 [Random Access], page 337.

port-has-set-port-position!? port
   Return #t is port supports set-port-position!.

call-with-port port proc
   Call proc, passing it port and closing port upon exit of proc. Return the return values of proc.

port-eof? input-port
   Equivalent to (eof-object? (lookahead-u8 input-port)).

standard-input-port
standard-output-port
standard-error-port
   Returns a fresh binary input port connected to standard input, or a binary output port connected to the standard output or standard error, respectively. Whether the port supports the port-position and set-port-position! operations is implementation-dependent.

current-input-port
current-output-port
current-error-port
   See Section 6.14.9 [Default Ports], page 340.

open-bytevector-input-port bv [transcoder]
open-bytevector-output-port [transcoder]
   See Section 6.14.10.2 [Bytevector Ports], page 344.

make-custom-binary-input-port id read! get-position
   set-position! close
make-custom-binary-output-port id write! get-position
   set-position! close
make-custom-binary-input/output-port id read! write!
   get-position set-position! close
   See Section 6.14.10.4 [Custom Ports], page 345.

get-u8 port
lookahead-u8 port
get-bytevector-n port count
get-bytevector-n! port bv start count
get-bytevector-some \textit{port} \hspace{1cm} \text{[Scheme Procedure]}
get-bytevector-all \textit{port} \hspace{1cm} \text{[Scheme Procedure]}
put-u8 \textit{port octet} \hspace{1cm} \text{[Scheme Procedure]}
put-bytevector \textit{port bv} \text{[start [count]]} \hspace{1cm} \text{[Scheme Procedure]}


get-char \textit{textual-input-port} \hspace{1cm} \text{[Scheme Procedure]}
lookahead-char \textit{textual-input-port} \hspace{1cm} \text{[Scheme Procedure]}
get-string-n \textit{textual-input-port count} \hspace{1cm} \text{[Scheme Procedure]}
get-string-n! \textit{textual-input-port string start count} \hspace{1cm} \text{[Scheme Procedure]}
get-string-all \textit{textual-input-port} \hspace{1cm} \text{[Scheme Procedure]}
get-line \textit{textual-input-port} \hspace{1cm} \text{[Scheme Procedure]}
put-char \textit{port char} \hspace{1cm} \text{[Scheme Procedure]}
put-string \textit{port string} \text{[start [count]]} \hspace{1cm} \text{[Scheme Procedure]}

See Section 6.14.4 [Textual I/O], page 333.

get-datum \textit{textual-input-port count} \hspace{1cm} \text{[Scheme Procedure]}

\begin{itemize}
  \item Reads an external representation from \textit{textual-input-port} and returns the datum it represents. The \textit{get-datum} procedure returns the next datum that can be parsed from the given \textit{textual-input-port}, updating \textit{textual-input-port} to point exactly past the end of the external representation of the object.
  \item Any \textit{interlexeme space} (comment or whitespace, see Section 6.18.1 [Scheme Syntax], page 379) in the input is first skipped. If an end of file occurs after the interlexeme space, the end-of-file object is returned.
  \item If a character inconsistent with an external representation is encountered in the input, an exception with condition types \&lexical and \&i/o-read is raised. Also, if the end of file is encountered after the beginning of an external representation, but the external representation is incomplete and therefore cannot be parsed, an exception with condition types \&lexical and \&i/o-read is raised.
\end{itemize}

put-datum \textit{textual-output-port datum} \hspace{1cm} \text{[Scheme Procedure]}

\textit{datum} should be a datum value. The \textit{put-datum} procedure writes an external representation of \textit{datum} to \textit{textual-output-port}. The specific external representation is implementation-dependent. However, whenever possible, an implementation should produce a representation for which \textit{get-datum}, when reading the representation, will return an object equal (in the sense of \texttt{equal?}) to \textit{datum}.

\textbf{Note:} Not all datums may allow producing an external representation for which \textit{get-datum} will produce an object that is equal to the original. Specifically, NaNs contained in \textit{datum} may make this impossible.

\textbf{Note:} The \textit{put-datum} procedure merely writes the external representation, but no trailing delimiter. If \textit{put-datum} is used to write several subsequent external representations to an output port, care should be taken to delimit them properly so they can be read back in by subsequent calls to \textit{get-datum}.

flush-output-port \textit{port} \hspace{1cm} \text{[Scheme Procedure]}

See Section 6.14.6 [Buffering], page 336, for documentation on \texttt{force-output}.
7.6.2.17 R6RS File Ports

The facilities described in this section are exported by the `(rnrs io ports)` module.

```scheme
buffer-mode buffer-mode-symbol
```
[Scheme Syntax]

`buffer-mode-symbol` must be a symbol whose name is one of `none`, `line`, and `block`.

The result is the corresponding symbol, and specifies the associated buffer mode. See Section 6.14.6 [Buffering], page 336, for a discussion of these different buffer modes.

To control the amount of buffering, use `setvbuf` instead. Note that only the name of `buffer-mode-symbol` is significant.

See Section 6.14.6 [Buffering], page 336, for a discussion of port buffering.

```scheme
buffer-mode? obj
```
[Scheme Procedure]

Returns `#t` if the argument is a valid buffer-mode symbol, and returns `#f` otherwise.

When opening a file, the various procedures accept a `file-options` object that encapsulates flags to specify how the file is to be opened. A `file-options` object is an enum-set (see Section 7.6.2.26 [rnrs enums], page 688) over the symbols constituting valid file options.

A `file-options` parameter name means that the corresponding argument must be a `file-options` object.

```scheme
file-options file-options-symbol ...
```
[Scheme Syntax]

Each `file-options-symbol` must be a symbol.

The `file-options` syntax returns a file-options object that encapsulates the specified options.

When supplied to an operation that opens a file for output, the file-options object returned by `(file-options)` specifies that the file is created if it does not exist and an exception with condition type `&i/o-file-already-exists` is raised if it does exist.

The following standard options can be included to modify the default behavior.

- `no-create` If the file does not already exist, it is not created; instead, an exception with condition type `&i/o-file-does-not-exist` is raised. If the file already exists, the exception with condition type `&i/o-file-already-exists` is not raised and the file is truncated to zero length.

- `no-fail` If the file already exists, the exception with condition type `&i/o-file-already-exists` is not raised, even if `no-create` is not included, and the file is truncated to zero length.

- `no-truncate` If the file already exists and the exception with condition type `&i/o-file-already-exists` has been inhibited by inclusion of `no-create` or `no-fail`, the file is not truncated, but the port’s current position is still set to the beginning of the file.

These options have no effect when a file is opened only for input. Symbols other than those listed above may be used as `file-options-symbols`; they have implementation-specific meaning, if any.

**Note:** Only the name of `file-options-symbol` is significant.
open-file-input-port filename

open-file-input-port filename file-options

open-file-input-port filename file-options buffer-mode

maybe-transcoder

maybe-transcoder must be either a transcoder or #f.

The open-file-input-port procedure returns an input port for the named file. The file-options and maybe-transcoder arguments are optional.

The file-options argument, which may determine various aspects of the returned port, defaults to the value of (file-options).

The buffer-mode argument, if supplied, must be one of the symbols that name a buffer mode. The buffer-mode argument defaults to block.

If maybe-transcoder is a transcoder, it becomes the transcoder associated with the returned port.

If maybe-transcoder is #f or absent, the port will be a binary port and will support the port-position and set-port-position! operations. Otherwise the port will be a textual port, and whether it supports the port-position and set-port-position! operations is implementation-dependent (and possibly transcoder-dependent).

open-file-output-port filename

open-file-output-port filename file-options

open-file-output-port filename file-options buffer-mode

maybe-transcoder

maybe-transcoder must be either a transcoder or #f.

The open-file-output-port procedure returns an output port for the named file.

The file-options argument, which may determine various aspects of the returned port, defaults to the value of (file-options).

The buffer-mode argument, if supplied, must be one of the symbols that name a buffer mode. The buffer-mode argument defaults to block.

If maybe-transcoder is a transcoder, it becomes the transcoder associated with the port.

If maybe-transcoder is #f or absent, the port will be a binary port and will support the port-position and set-port-position! operations. Otherwise the port will be a textual port, and whether it supports the port-position and set-port-position! operations is implementation-dependent (and possibly transcoder-dependent).

7.6.2.18 rnrs io simple

The (rnrs io simple (6)) library provides convenience functions for performing textual I/O on ports. This library also exports all of the condition types and associated procedures described in (see Section 7.6.2.14 [R6RS I/O Conditions], page 668). In the context of this section, when stating that a procedure behaves “identically” to the corresponding procedure in Guile’s core library, this is modulo the behavior wrt. conditions: such procedures raise the appropriate R6RS conditions in case of error, but otherwise behave identically.
**Note:** There are still known issues regarding condition-correctness; some errors may still be thrown as native Guile exceptions instead of the appropriate R6RS conditions.

**eof-object** [Scheme Procedure]
**eof-object? obj** [Scheme Procedure]
These procedures are identical to the ones provided by the (rnrs io ports (6)) library. See Section 7.6.2.16 [rnrs io ports], page 672, for documentation.

**input-port? obj** [Scheme Procedure]
**output-port? obj** [Scheme Procedure]
These procedures are identical to the ones provided by Guile’s core library. See Section 6.14.1 [Ports], page 328, for documentation.

**call-with-input-file filename proc** [Scheme Procedure]
**call-with-output-file filename proc** [Scheme Procedure]
**open-input-file filename** [Scheme Procedure]
**open-output-file filename** [Scheme Procedure]
**with-input-from-file filename thunk** [Scheme Procedure]
**with-output-to-file filename thunk** [Scheme Procedure]
These procedures are identical to the ones provided by Guile’s core library. See Section 6.14.10.1 [File Ports], page 341, for documentation.

**close-input-port input-port** [Scheme Procedure]
**close-output-port output-port** [Scheme Procedure]
Closes the given input-port or output-port. These are legacy interfaces; just use close-port.

**peek-char** [Scheme Procedure]
**peek-char textual-input-port** [Scheme Procedure]
**read-char** [Scheme Procedure]
**read-char textual-input-port** [Scheme Procedure]
These procedures are identical to the ones provided by Guile’s core library. See Section 6.14.11 [Venerable Port Interfaces], page 347, for documentation.

**read textual-input-port** [Scheme Procedure]
This procedure is identical to the one provided by Guile’s core library. See Section 6.18.2 [Scheme Read], page 382, for documentation.

**display obj** [Scheme Procedure]
**display obj textual-output-port** [Scheme Procedure]
**newline** [Scheme Procedure]
**newline textual-output-port** [Scheme Procedure]
**write obj** [Scheme Procedure]
**write obj textual-output-port** [Scheme Procedure]
**write-char char** [Scheme Procedure]
write-char char textual-output-port  
  These procedures are identical to the ones provided by Guile’s core library. See Section 6.14.11 [Venerable Port Interfaces], page 347, and See Section 6.18.3 [Scheme Write], page 383, for documentation.

7.6.2.19 rnrs files

The (rnrs files (6)) library provides the file-exists? and delete-file procedures, which test for the existence of a file and allow the deletion of files from the file system, respectively.

These procedures are identical to the ones provided by Guile’s core library. See Section 7.2.3 [File System], page 500, for documentation.

7.6.2.20 rnrs programs

The (rnrs programs (6)) library provides procedures for process management and introspection.

command-line  
  This procedure is identical to the one provided by Guile’s core library. See Section 7.2.6 [Runtime Environment], page 512, for documentation.

exit [status]  
  This procedure is identical to the one provided by Guile’s core library. See Section 7.2.7 [Processes], page 514, for documentation.

7.6.2.21 rnrs arithmetic fixnums

The (rnrs arithmetic fixnums (6)) library provides procedures for performing arithmetic operations on an implementation-dependent range of exact integer values, which R6RS refers to as fixnums. In Guile, the size of a fixnum is determined by the size of the SCM type; a single SCM struct is guaranteed to be able to hold an entire fixnum, making fixnum computations particularly efficient—(see Section 6.3 [The SCM Type], page 100). On 32-bit systems, the most negative and most positive fixnum values are, respectively, -536870912 and 536870911.

Unless otherwise specified, all of the procedures below take fixnums as arguments, and will raise an &assertion condition if passed a non-fixnum argument or an &implementation-restriction condition if their result is not itself a fixnum.

fixnum? obj  
  Returns #t if obj is a fixnum, #f otherwise.

fixnum-width  
leastr-fixnum  
greatest-fixnum  
  These procedures return, respectively, the maximum number of bits necessary to represent a fixnum value in Guile, the minimum fixnum value, and the maximum fixnum value.

fx=? fx1 fx2 fx3 ...  
fx>>? fx1 fx2 fx3 ...
These procedures return #t if their fixnum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; #f otherwise.

These numerical predicates return #t if fx is, respectively, zero, greater than zero, less than zero, odd, or even; #f otherwise.

These procedures return the maximum or minimum of their arguments.

These procedures return the sum or product of their arguments.

Returns the difference of fx1 and fx2, or the negation of fx, if called with a single argument.

An &assertion condition is raised if the result is not itself a fixnum.

These procedures implement number-theoretic division on fixnums; See ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩, for a description of their semantics.

Returns the two fixnum results of the following computation:

\[
\begin{align*}
\text{let*} & \quad \left(\text{s0 (mod0 s (expt 2 (fixnum-width)))} \right) \\
& \quad \left(\text{s1 (div0 s (expt 2 (fixnum-width)))} \right) \\
& \quad \left(\text{values s0 s1} \right)
\end{align*}
\]

Returns the two fixnum results of the following computation:

\[
\begin{align*}
\text{let*} & \quad \left(\text{d (- fx1 fx2 fx3)} \right)
\end{align*}
\]
(d0 (mod0 d (expt 2 (fixnum-width))))
(d1 (div0 d (expt 2 (fixnum-width)))))
(values d0 d1))

\(\texttt{fx*/carry \, fx1 \, fx2 \, fx3}\) [Scheme Procedure]

Returns the two fixnum results of the following computation:
\[
\begin{align*}
\text{(let* ((s (+ (* fx1 fx2) fx3)))} \\
\text{(s0 (mod0 s (expt 2 (fixnum-width)))))} \\
\text{(s1 (div0 s (expt 2 (fixnum-width)))))} \\
\end{align*}
\]
(values s0 s1))

\(\texttt{fxnot \, fx}\) [Scheme Procedure]

\(\texttt{fxand \, fx1 \, ...}\) [Scheme Procedure]

\(\texttt{fxior \, fx1 \, ...}\) [Scheme Procedure]

\(\texttt{fxxor \, fx1 \, ...}\) [Scheme Procedure]

These procedures are identical to the \texttt{lognot}, \texttt{logand}, \texttt{logior}, and \texttt{logxor} procedures provided by Guile’s core library. See Section 6.6.2.13 [Bitwise Operations], page 125, for documentation.

\(\texttt{fxif \, fx1 \, fx2 \, fx3}\) [Scheme Procedure]

Returns the bitwise “if” of its fixnum arguments. The bit at position \(i\) in the return value will be the \(i\)th bit from \(fx2\) if the \(i\)th bit of \(fx1\) is 1, the \(i\)th bit from \(fx3\).

\(\texttt{fxbit-count \, fx}\) [Scheme Procedure]

Returns the number of 1 bits in the two’s complement representation of \(fx\).

\(\texttt{fxlength \, fx}\) [Scheme Procedure]

Returns the number of bits necessary to represent \(fx\).

\(\texttt{fxfirst-bit-set \, fx}\) [Scheme Procedure]

Returns the index of the least significant 1 bit in the two’s complement representation of \(fx\).

\(\texttt{fxbit-set? \, fx1 \, fx2}\) [Scheme Procedure]

Returns \#t if the \(fx2\)th bit in the two’s complement representation of \(fx1\) is 1, \#f otherwise.

\(\texttt{fxcopy-bit \, fx1 \, fx2 \, fx3}\) [Scheme Procedure]

Returns the result of setting the \(fx2\)th bit of \(fx1\) to the \(fx2\)th bit of \(fx3\).

\(\texttt{fxbit-field \, fx1 \, fx2 \, fx3}\) [Scheme Procedure]

Returns the integer representation of the contiguous sequence of bits in \(fx1\) that starts at position \(fx2\) (inclusive) and ends at position \(fx3\) (exclusive).

\(\texttt{fxcopy-bit-field \, fx1 \, fx2 \, fx3 \, fx4}\) [Scheme Procedure]

Returns the result of replacing the bit field in \(fx1\) with start and end positions \(fx2\) and \(fx3\) with the corresponding bit field from \(fx4\).

\(\texttt{fxarithmetic-shift \, fx1 \, fx2}\) [Scheme Procedure]

\(\texttt{fxarithmetic-shift-left \, fx1 \, fx2}\) [Scheme Procedure]

\(\texttt{fxarithmetic-shift-right \, fx1 \, fx2}\) [Scheme Procedure]

Returns the result of shifting the bits of \(fx1\) right or left by the \(fx2\) positions. \(\texttt{fxarithmetic-shift}\) is identical to \(\texttt{fxarithmetic-shift-left}\).
**fxrotate-bit-field**  
```scheme
fx1 fx2 fx3 fx4
```
Returns the result of cyclically permuting the bit field in `fx1` with start and end positions `fx2` and `fx3` by `fx4` bits in the direction of more significant bits.

**fxreverse-bit-field**  
```scheme
fx1 fx2 fx3
```
Returns the result of reversing the order of the bits of `fx1` between position `fx2` (inclusive) and position `fx3` (exclusive).

### 7.6.2.22 rnrs arithmetic flonums
The `(rnrs arithmetic flonums (6))` library provides procedures for performing arithmetic operations on inexact representations of real numbers, which R6RS refers to as flonums.

Unless otherwise specified, all of the procedures below take flonums as arguments, and will raise an `&assertion` condition if passed a non-flonum argument.

**flonum? obj**  
```scheme
Returns #t if obj is a flonum, #f otherwise.
```

**real->flonum x**  
```scheme
Returns the flonum that is numerically closest to the real number x.
```

**fl=? fl1 fl2 fl3 ...**  
**fl<? fl1 fl2 fl3 ...**  
**fl<=? fl1 fl2 fl3 ...**  
**fl>? fl1 fl2 fl3 ...**  
**fl>=? fl1 fl2 fl3 ...**

These procedures return #t if their flonum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; #f otherwise.

**flinteger? fl**  
**flzero? fl**  
**flpositive? fl**  
**flnegative? fl**  
**flodd? fl**  
**fleven? fl**

These numerical predicates return #t if `fl` is, respectively, an integer, zero, greater than zero, less than zero, odd, even, #f otherwise. In the case of `flodd?` and `fleven?`, `fl` must be an integer-valued flonum.

**flfinite? fl**  
**flinfinite? fl**  
**flnan? fl**

These numerical predicates return #t if `fl` is, respectively, not infinite, infinite, or a NaN value.

**flmax fl1 fl2 ...**  
**flmin fl1 fl2 ...**

These procedures return the maximum or minimum of their arguments.
Chapter 7: Guile Modules

\[ \text{fl+ } \text{fl} \ldots \]  
\[ \text{fl* } \text{fl} \ldots \]  
These procedures return the sum or product of their arguments.

\[ \text{fl- } \text{fl1 } \text{fl2 } \ldots \]  
\[ \text{fl- } \text{fl} \]  
\[ \text{fl/ } \text{fl1 } \text{fl2 } \ldots \]  
\[ \text{fl/ } \text{fl} \]  
These procedures return, respectively, the difference or quotient of their arguments when called with two arguments; when called with a single argument, they return the additive or multiplicative inverse of \( \text{fl} \).

\[ \text{flabs } \text{fl} \]  
Returns the absolute value of \( \text{fl} \).

\[ \text{fldiv-and-mod } \text{fl1 } \text{fl2} \]  
\[ \text{fldiv } \text{fl1 } \text{fl2} \]  
\[ \text{fldmod } \text{fl1 } \text{fl2} \]  
\[ \text{fldiv0-and-mod0 } \text{fl1 } \text{fl2} \]  
\[ \text{fldiv0 } \text{fl1 } \text{fl2} \]  
\[ \text{flmod0 } \text{fl1 } \text{fl2} \]  
These procedures implement number-theoretic division on flonums; See (undefined) \((\text{rnrs base})\), page (undefined), for a description for their semantics.

\[ \text{flnumerator } \text{fl} \]  
\[ \text{fldenominator } \text{fl} \]  
These procedures return the numerator or denominator of \( \text{fl} \) as a flonum.

\[ \text{flfloor } \text{fl1} \]  
\[ \text{flceiling } \text{fl} \]  
\[ \text{fltruncate } \text{fl} \]  
\[ \text{flround } \text{fl} \]  
These procedures are identical to the \text{floor}, \text{ceiling}, \text{truncate}, and \text{round} procedures provided by Guile’s core library. See Section 6.6.2.11 [Arithmetic], page 119, for documentation.

\[ \text{flexp } \text{fl} \]  
\[ \text{fllog } \text{fl} \]  
\[ \text{fllog } \text{fl1 } \text{fl2} \]  
\[ \text{flsin } \text{fl} \]  
\[ \text{flcos } \text{fl} \]  
\[ \text{fltan } \text{fl} \]  
\[ \text{flasin } \text{fl} \]  
\[ \text{flacos } \text{fl} \]  
\[ \text{flatan } \text{fl} \]  
\[ \text{flatan } \text{fl1 } \text{fl2} \]  
These procedures, which compute the usual transcendental functions, are the flonum variants of the procedures provided by the R6RS base library (see (undefined) \((\text{rnrs base})\), page (undefined)).
flsqrt fl

Returns the square root of fl. If fl is −0.0, -0.0 is returned; for other negative values, a NaN value is returned.

flexpt fl1 fl2

Returns the value of fl1 raised to the power of fl2.

The following condition types are provided to allow Scheme implementations that do not support infinities or NaN values to indicate that a computation resulted in such a value. Guile supports both of these, so these conditions will never be raised by Guile’s standard libraries implementation.

&no-infinities

[Condition Type]

make-no-infinities-violation obj

[Scheme Procedure]

no-infinities-violation? obj

[Scheme Procedure]

A condition type indicating that a computation resulted in an infinite value on a Scheme implementation incapable of representing infinities.

&no-nans

[Condition Type]

make-no-nans-violation obj

[Scheme Procedure]

no-nans-violation? obj

[Scheme Procedure]

A condition type indicating that a computation resulted in a NaN value on a Scheme implementation incapable of representing NaNs.

fixnum->flonum fx

[Scheme Procedure]

Returns the flonum that is numerically closest to the fixnum fx.

7.6.2.23 rnrs arithmetic bitwise

The (rnrs arithmetic bitwise (6)) library provides procedures for performing bitwise arithmetic operations on the two’s complement representations of fixnums.

This library and the procedures it exports share functionality with SRFI-60, which provides support for bitwise manipulation of integers (see Section 7.5.34 [SRFI-60], page 642).

bitwise-not ei

[Scheme Procedure]

bitwise-and ei1 ...

[Scheme Procedure]

bitwise-ior ei1 ...

[Scheme Procedure]

bitwise-xor ei1 ...

[Scheme Procedure]

These procedures are identical to the lognot, logand, logior, and logxor procedures provided by Guile’s core library. See Section 6.6.2.13 [Bitwise Operations], page 125, for documentation.

bitwise-if ei1 ei2 ei3

[Scheme Procedure]

Returns the bitwise “if” of its arguments. The bit at position i in the return value will be the ith bit from ei2 if the ith bit of ei1 is 1, the ith bit from ei3.

bitwise-bit-count ei

[Scheme Procedure]

Returns the number of 1 bits in the two’s complement representation of ei.

bitwise-length ei

[Scheme Procedure]

Returns the number of bits necessary to represent ei.
bitwise-first-bit-set ei
   Returns the index of the least significant 1 bit in the two's complement representation of ei.

bitwise-bit-set? ei1 ei2
   Returns #t if the ei2th bit in the two's complement representation of ei1 is 1, #f otherwise.

bitwise-copy-bit ei1 ei2 ei3
   Returns the result of setting the ei2th bit of ei1 to the ei2th bit of ei3.

bitwise-copy-bit-field ei1 ei2 ei3 ei4
   Returns the result of replacing the bit field in ei1 with start and end positions ei2 and ei3 with the corresponding bit field from ei4.

bitwise-arithmetic-shift ei1 ei2
   Returns the result of shifting the bits of ei1 right or left by the ei2 positions.

bitwise-arithmetic-shift-left ei1 ei2
   Returns the integer representation of the contiguous sequence of bits in ei1 that starts at position ei2 (inclusive) and ends at position ei3 (exclusive).

bitwise-arithmetic-shift-right ei1 ei2
   Returns the result of cyclically permuting the bit field in ei1 with start and end positions ei2 and ei3 by ei4 bits in the direction of more significant bits.

bitwise-reverse-bit-field ei1 ei2 ei3
   Returns the result of reversing the order of the bits of ei1 between position ei2 (inclusive) and position ei3 (exclusive).

7.6.2.24 rnrs syntax-case
   The (rnrs syntax-case (6)) library provides access to the syntax-case system for writing hygienic macros. With one exception, all of the forms and procedures exported by this library are “re-exports” of Guile’s native support for syntax-case; See Section 6.10.3 [Syntax Case], page 267, for documentation, examples, and rationale.

make-variable-transformer proc
   Creates a new variable transformer out of proc, a procedure that takes a syntax object as input and returns a syntax object. If an identifier to which the result of this procedure is bound appears on the left-hand side of a set! expression, proc will be called with a syntax object representing the entire set! expression, and its return value will replace that set! expression.

syntax-case expression (literal ...) clause ...
   The syntax-case pattern matching form.
syntax template
quasisyntax template
unsyntax template
unsyntax-splicing template

These forms allow references to be made in the body of a syntax-case output expression subform to datum and non-datum values. They are identical to the forms provided by Guile’s core library; See Section 6.10.3 [Syntax Case], page 267, for documentation.

identifier? obj
bound-identifier=? id1 id2
free-identifier=? id1 id2

These predicate procedures operate on syntax objects representing Scheme identifiers. identifier? returns #t if obj represents an identifier, #f otherwise. bound-identifier=? returns #t if and only if a binding for id1 would capture a reference to id2 in the transformer’s output, or vice-versa. free-identifier=? returns #t if and only id1 and id2 would refer to the same binding in the output of the transformer, independent of any bindings introduced by the transformer.

generate-temporaries l

Returns a list, of the same length as l, which must be a list or a syntax object representing a list, of globally unique symbols.

syntax->datum syntax-object
datum->syntax template-id datum

These procedures convert wrapped syntax objects to and from Scheme datum values. The syntax object returned by datum->syntax shares contextual information with the syntax object template-id.

syntax-violation whom message form

Constructs a new compound condition that includes the following simple conditions:

- If whom is not #f, a &who condition with the whom as its field
- A &message condition with the specified message
- A &syntax condition with the specified form and optional subform fields

7.6.2.25 rnrs hashtables

The (rnrs hashtables (6)) library provides structures and procedures for creating and accessing hash tables. The hash tables API defined by R6RS is substantially similar to both Guile’s native hash tables implementation as well as the one provided by SRFI-69; See Section 6.6.22 [Hash Tables], page 237, and Section 7.5.39 [SRFI-69], page 644, respectively. Note that you can write portable R6RS library code that manipulates SRFI-69 hash tables (by importing the (srfi :69) library); however, hash tables created by one API cannot be used by another.

Like SRFI-69 hash tables—and unlike Guile’s native ones—R6RS hash tables associate hash and equality functions with a hash table at the time of its creation. Additionally, R6RS allows for the creation (via hashtable-copy; see below) of immutable hash tables.
**make-eq-hashtable**

Returns a new hash table that uses eq? to compare keys and Guile’s hashq procedure as a hash function. If k is given, it specifies the initial capacity of the hash table.

**make-eqv-hashtable**

Returns a new hash table that uses eqv? to compare keys and Guile’s hashv procedure as a hash function. If k is given, it specifies the initial capacity of the hash table.

**make-hashtable** hash-function equiv

Returns a new hash table that uses equiv to compare keys and hash-function as a hash function. equiv must be a procedure that accepts two arguments and returns a true value if they are equivalent, #f otherwise; hash-function must be a procedure that accepts one argument and returns a non-negative integer.

If k is given, it specifies the initial capacity of the hash table.

**hashtable?** obj

Returns #t if obj is an R6RS hash table, #f otherwise.

**hashtable-size** hashtable

Returns the number of keys currently in the hash table hashtable.

**hashtable-ref** hashtable key default

Returns the value associated with key in the hash table hashtable, or default if none is found.

**hashtable-set!** hashtable key obj

Associates the key key with the value obj in the hash table hashtable, and returns an unspecified value. An &assertion condition is raised if hashtable is immutable.

**hashtable-delete!** hashtable key

Removes any association found for the key key in the hash table hashtable, and returns an unspecified value. An &assertion condition is raised if hashtable is immutable.

**hashtable-contains?** hashtable key

Returns #t if the hash table hashtable contains an association for the key key, #f otherwise.

**hashtable-update!** hashtable key proc default

Associates with key in the hash table hashtable the result of calling proc, which must be a procedure that takes one argument, on the value currently associated key in hashtable—or on default if no such association exists. An &assertion condition is raised if hashtable is immutable.

**hashtable-copy** hashtable

Returns a copy of the hash table hashtable. If the optional argument mutable is provided and is a true value, the new hash table will be mutable.
hashtable-clear! hashtable  [Scheme Procedure]
hashtable-clear! hashtable k  [Scheme Procedure]
   Removes all of the associations from the hash table hashtable. The optional argument
   k, which specifies a new capacity for the hash table, is accepted by Guile's (rnrs
   hashtables) implementation, but is ignored.

hashtable-keys hashtable  [Scheme Procedure]
   Returns a vector of the keys with associations in the hash table hashtable, in an
   unspecified order.

hashtable-entries hashtable  [Scheme Procedure]
   Return two values—a vector of the keys with associations in the hash table hashtable,
   and a vector of the values to which these keys are mapped, in corresponding but
   unspecified order.

hashtable-equivalence-function hashtable  [Scheme Procedure]
   Returns the equivalence predicated use by hashtable. This procedure returns eq? and
   eqv?, respectively, for hash tables created by make-eq-hashtable and make-eqv-
   hashtable.

hashtable-hash-function hashtable  [Scheme Procedure]
   Returns the hash function used by hashtable. For hash tables created by make-eq-
   hashtable or make-eqv-hashtable, #f is returned.

hashtable-mutable? hashtable  [Scheme Procedure]
   Returns #t if hashtable is mutable, #f otherwise.

A number of hash functions are provided for convenience:

equal-hash obj  [Scheme Procedure]
   Returns an integer hash value for obj, based on its structure and current contents.
   This hash function is suitable for use with equal? as an equivalence function.

string-hash string  [Scheme Procedure]
symbol-hash symbol  [Scheme Procedure]
   These procedures are identical to the ones provided by Guile’s core library. See
   Section 6.6.22.2 [Hash Table Reference], page 239, for documentation.

string-ci-hash string  [Scheme Procedure]
   Returns an integer hash value for string based on its contents, ignoring case. This
   hash function is suitable for use with string-ci=? as an equivalence function.

7.6.2.26 rnrs enums

The (rnrs enums (6)) library provides structures and procedures for working with enu-
merable sets of symbols. Guile’s implementation defines an enum-set record type that
encapsulates a finite set of distinct symbols, the universe, and a subset of these symbols,
which define the enumeration set.

The SRFI-1 list library provides a number of procedures for performing set opera-
tions on lists; Guile’s (rnrs enums) implementation makes use of several of them. See
Section 7.5.3.10 [SRFI-1 Set Operations], page 589, for more information.
make-enumeration symbol-list
   [Scheme Procedure]
   Returns a new enum-set whose universe and enumeration set are both equal to symbol-list, a list of symbols.

enum-set-universe enum-set
   [Scheme Procedure]
   Returns an enum-set representing the universe of enum-set, an enum-set.

enum-set-indexer enum-set
   [Scheme Procedure]
   Returns a procedure that takes a single argument and returns the zero-indexed position of that argument in the universe of enum-set, or #f if its argument is not a member of that universe.

enum-set-constructor enum-set
   [Scheme Procedure]
   Returns a procedure that takes a single argument, a list of symbols from the universe of enum-set, an enum-set, and returns a new enum-set with the same universe that represents a subset containing the specified symbols.

enum-set->list enum-set
   [Scheme Procedure]
   Returns a list containing the symbols of the set represented by enum-set, an enum-set, in the order that they appear in the universe of enum-set.

enum-set-member? symbol enum-set
enum-set-subset? enum-set1 enum-set2
enum-set= enum-set1 enum-set2
   [Scheme Procedure]
   These procedures test for membership of symbols and enum-sets in other enum-sets.
   enum-set-member? returns #t if and only if symbol is a member of the subset specified by enum-set. enum-set-subset? returns #t if and only if the universe of enum-set1 is a subset of the universe of enum-set2 and every symbol in enum-set1 is present in enum-set2. enum-set= returns #t if and only if enum-set1 is a subset, as per enum-set-subset? of enum-set2 and vice versa.

enum-set-union enum-set1 enum-set2
enum-set-intersection enum-set1 enum-set2
enum-set-difference enum-set1 enum-set2
   [Scheme Procedure]
   These procedures return, respectively, the union, intersection, and difference of their enum-set arguments.

enum-set-complement enum-set
   [Scheme Procedure]
   Returns enum-set’s complement (an enum-set), with regard to its universe.

enum-set-projection enum-set1 enum-set2
   [Scheme Procedure]
   Returns the projection of the enum-set enum-set1 onto the universe of the enum-set enum-set2.

define-enumeration type-name (symbol ...)
   [Scheme Syntax]
   constructor-syntax
   Evaluates to two new definitions: A constructor bound to constructor-syntax that behaves similarly to constructors created by enum-set-constructor, above, and creates new enum-sets in the universe specified by (symbol ...); and a “predicate macro” bound to type-name, which has the following form:

   (type-name sym)
If \( \text{sym} \) is a member of the universe specified by the \textit{symbols} above, this form evaluates to \( \text{sym} \). Otherwise, a \&\textit{syntax} condition is raised.

### 7.6.2.27 rnrs

The \( \text{(rnrs (6))} \) library is a composite of all of the other R6RS standard libraries—it imports and re-exports all of their exported procedures and syntactic forms—with the exception of the following libraries:

- \( \text{(rnrs eval (6))} \)
- \( \text{(rnrs mutable-pairs (6))} \)
- \( \text{(rnrs mutable-strings (6))} \)
- \( \text{(rnrs r5rs (6))} \)

### 7.6.2.28 rnrs eval

The \( \text{(rnrs eval (6))} \) library provides procedures for performing “on-the-fly” evaluation of expressions.

\[
\text{eval} \quad \text{expression} \quad \text{environment} \quad \text{[Scheme Procedure]}
\]

Evaluates \text{expression}, which must be a datum representation of a valid Scheme expression, in the environment specified by \text{environment}. This procedure is identical to the one provided by Guile’s code library; See Section 6.18.4 [Fly Evaluation], page 384, for documentation.

\[
\text{environment} \quad \text{import-spec} \quad \text{...} \quad \text{[Scheme Procedure]}
\]

Constructs and returns a new environment based on the specified \text{import-specs}, which must be datum representations of the import specifications used with the \text{import} form. See Section 6.20.6 [R6RS Libraries], page 414, for documentation.

### 7.6.2.29 rnrs mutable-pairs

The \( \text{(rnrs mutable-pairs (6))} \) library provides the \text{set-car!} and \text{set-cdr!} procedures, which allow the \text{car} and \text{cdr} fields of a pair to be modified.

These procedures are identical to the ones provide by Guile’s core library. See Section 6.6.8 [Pairs], page 178, for documentation. All pairs in Guile are mutable; consequently, these procedures will never throw the \&\textit{assertion} condition described in the R6RS libraries specification.

### 7.6.2.30 rnrs mutable-strings

The \( \text{(rnrs mutable-strings (6))} \) library provides the \text{string-set!} and \text{string-fill!} procedures, which allow the content of strings to be modified “in-place.”

These procedures are identical to the ones provided by Guile’s core library. See Section 6.6.5.6 [String Modification], page 147, for documentation. All strings in Guile are mutable; consequently, these procedures will never throw the \&\textit{assertion} condition described in the R6RS libraries specification.

### 7.6.2.31 rnrs r5rs

The \( \text{(rnrs r5rs (6))} \) library exports bindings for some procedures present in R5RS but omitted from the R6RS base library specification.
exact->inexact z  
exact->inexact z  

These procedures are identical to the ones provided by Guile’s core library. See Section 6.6.2.5 [Exactness], page 113, for documentation.

quotient n1 n2  
remainder n1 n2  
modulo n1 n2  

These procedures are identical to the ones provided by Guile’s core library. See Section 6.6.2.7 [Integer Operations], page 116, for documentation.

delay expr  
force promise  

The delay form and the force procedure are identical to their counterparts in Guile’s core library. See Section 6.18.9 [Delayed Evaluation], page 393, for documentation.

null-environment n  
scheme-report-environment n  

These procedures are identical to the ones provided by the (ice-9 r5rs) Guile module. See Section 6.20.11 [Environments], page 421, for documentation.

7.7 Pattern Matching

The (ice-9 match) module provides a pattern matcher, written by Alex Shinn, and compatible with Andrew K. Wright’s pattern matcher found in many Scheme implementations.

A pattern matcher can match an object against several patterns and extract the elements that make it up. Patterns can represent any Scheme object: lists, strings, symbols, records, etc. They can optionally contain pattern variables. When a matching pattern is found, an expression associated with the pattern is evaluated, optionally with all pattern variables bound to the corresponding elements of the object:

(let ((l '(hello (world))))  
  (match l  
    (('hello (who))  
      who)  
    ;; <- the input object  
    ;; <- the pattern  
    ;; <- the expression evaluated upon matching  
  )  
⇒ world  

In this example, list l matches the pattern ('hello (who)), because it is a two-element list whose first element is the symbol hello and whose second element is a one-element list. Here who is a pattern variable. match, the pattern matcher, locally binds who to the value contained in this one-element list—i.e., the symbol world. An error would be raised if l did not match the pattern.

The same object can be matched against a simpler pattern:

(let ((l '(hello (world))))  
  (match l  
    ((x y)  
      (values x y)))  
⇒ hello  
⇒ (world)
Here pattern \((x \ y)\) matches any two-element list, regardless of the types of these elements. Pattern variables \(x\) and \(y\) are bound to, respectively, the first and second element of \(l\).

Patterns can be composed, and nested. For instance, \(\ldots\) (ellipsis) means that the previous pattern may be matched zero or more times in a list:

\[
\text{(match } lst \\
\quad ( ((\text{heads} \ \text{tails} \ \ldots) \ \ldots) \\
\quad \text{heads}) )
\]

This expression returns the first element of each list within \(lst\). For proper lists of proper lists, it is equivalent to \((\text{map car } lst)\). However, it performs additional checks to make sure that \(lst\) and the lists therein are proper lists, as prescribed by the pattern, raising an error if they are not.

Compared to hand-written code, pattern matching noticeably improves clarity and conciseness—no need to resort to series of \text{car} and \text{cdr} calls when matching lists, for instance. It also improves robustness, by making sure the input \textit{completely} matches the pattern—conversely, hand-written code often trades robustness for conciseness. And of course, \text{match} is a macro, and the code it expands to is just as efficient as equivalent hand-written code.

The pattern matcher is defined as follows:

\[
\text{match } \text{exp} \ \text{clause1 clause2 } \ldots \quad \text{[Scheme Syntax]} \\
\text{Match object } \text{exp} \text{ against the patterns in clause1 clause2 } \ldots \text{ in the order in which they appear. Return the value produced by the first matching clause. If no clause matches, throw an exception with key match-error.}
\]

Each clause has the form \((\text{pattern body1 body2 } \ldots)\). Each \text{pattern} must follow the syntax described below. Each \text{body} is an arbitrary Scheme expression, possibly referring to pattern variables of \text{pattern}.

The syntax and interpretation of patterns is as follows:

\[
\begin{align*}
\text{patterns:} & \quad \text{matches:} \\
\text{pat ::= identifier} & \quad \text{anything, and binds identifier} \\
| \_ & \quad \text{anything} \\
| () & \quad \text{the empty list} \\
| \#t & \quad \text{\#t} \\
| \#f & \quad \text{\#f} \\
| \text{string} & \quad \text{a string} \\
| \text{number} & \quad \text{a number} \\
| \text{character} & \quad \text{a character} \\
| \text{'sexp} & \quad \text{an s-expression} \\
| \text{'symbol} & \quad \text{a symbol (special case of s-expr)} \\
| (\text{pat}_1 \ldots \text{pat}_n) & \quad \text{list of } n \text{ elements} \\
| (\text{pat}_1 \ldots \text{pat}_n . \text{pat}_{\{n+1\}}) & \quad \text{list of } n \text{ or more} \\
| (\text{pat}_1 \ldots \text{pat}_n \text{ pat}_{n+1} \text{ ooo}) & \quad \text{list of } n \text{ or more, each element of remainder must match } \text{pat}_{n+1} \\
| \#(\text{pat}_1 \ldots \text{pat}_n) & \quad \text{vector of } n \text{ elements}
\end{align*}
\]
Chapter 7: Guile Modules

| #(pat_1 ... pat_n pat_n+1 ooo) vector of n or more, each element | vector of n or more, each element of remainder must match pat_n+1 |
| #&pat | box |
| ($ record-name pat_1 ... pat_n) a record |
| (= field pat) a ‘‘field’’ of an object |
| (and pat_1 ... pat_n) if all of pat_1 thru pat_n match |
| (or pat_1 ... pat_n) if any of pat_1 thru pat_n match |
| (not pat_1 ... pat_n) if all pat_1 thru pat_n don’t match |
| (? predicate pat_1 ... pat_n) if predicate true and all of pat_1 thru pat_n match |
| (set! identifier) anything, and binds setter |
| (get! identifier) anything, and binds getter |
| ‘qp | a quasi-pattern |
| (identifier *** pat) matches pat in a tree and binds identifier to the path leading to the object that matches pat |

ooo ::= ... zero or more |
| ___ zero or more |
| ..1 1 or more |

quasi-patterns: matches:

qp ::= () the empty list |
| #t #t |
| #f #f |
| string a string |
| number a number |
| character a character |
| identifier a symbol |
| (qp_1 ... qp_n) list of n elements |
| (qp_1 ... qp_n . qp_{n+1}) list of n or more |
| (qp_1 ... qp_n qp_n+1 ooo) list of n or more, each element of remainder must match qp_n+1 |
| #(qp_1 ... qp_n) vector of n elements |
| #(qp_1 ... qp_n qp_n+1 ooo) vector of n or more, each element of remainder must match qp_n+1 |
| #&qp | box |
| ,pat a pattern |
| ,@pat a pattern |

The names quote, quasiquote, unquote, unquote-splicing, ?, _, $, and, or, not, set!, get!, ..., and ___ cannot be used as pattern variables.

Here is a more complex example:

(use-modules (srfi srfi-9))

(let ()
(define-record-type person
  (make-person name friends)
  person?
  (name person-name)
  (friends person-friends))

(letrec ((alice (make-person "Alice" (delay (list bob))))
  (bob (make-person "Bob" (delay (list alice)))))
  (match alice
    (($ person name (= force (($ person "Bob"))))
      (list 'friend-of-bob name))
    (_ #f)))
⇒ (friend-of-bob "Alice")

Here the $ pattern is used to match a SRFI-9 record of type `person` containing two or more slots. The value of the first slot is bound to `name`. The = pattern is used to apply `force` on the second slot, and then checking that the result matches the given pattern. In other words, the complete pattern matches any `person` whose second slot is a promise that evaluates to a one-element list containing a `person` whose first slot is "Bob".

The (ice-9 match) module also provides the following convenient syntactic sugar macros wrapping around `match`.

**match-lambda**

```
match-lambda clause1 clause2 ... [Scheme Syntax]
Create a procedure of one argument that matches its argument against each clause,
and returns the result of evaluating the corresponding expressions.

(match-lambda clause1 clause2 ...)
≡
  (lambda (arg) (match arg clause1 clause2 ...))
```

```
((match-lambda
  (('hello (who))
   who))
  '(hello (world)))
⇒ world
```

**match-lambda***

```
match-lambda* clause1 clause2 ... [Scheme Syntax]
Create a procedure of any number of arguments that matches its argument list against
each clause, and returns the result of evaluating the corresponding expressions.

(match-lambda* clause1 clause2 ...)
≡
  (lambda args (match args clause1 clause2 ...))
```

```
((match-lambda*
  (('hello (who))
   who))
 'hello '(world))
⇒ world
```
**match-let** ((pattern expression) ...) body

Match each pattern to the corresponding expression, and evaluate the body with all
matched variables in scope. Raise an error if any of the expressions fail to match.
**match-let** is analogous to named let and can also be used for recursive functions
which match on their arguments as in **match-lambda**.

```scheme
(match-let (((x y) (list 1 2))
            ((a b) (list 3 4)))
  (list a b x y))
⇒
(3 4 1 2)
```

**match-let** variable ((pattern init) ...) body

Similar to **match-let**, but analogously to named let, locally bind VARIABLE to a
new procedure which accepts as many arguments as there are INIT expressions. The
procedure is initially applied to the results of evaluating the INIT expressions. When
called, the procedure matches each argument against the corresponding PATTERN,
and returns the result(s) of evaluating the BODY expressions. See Section 6.13.4
[while do], page 301, for more on named let.

```scheme
(match-let* (((x y) (list 1 2))
              ((a b) (list x 4)))
  (list a b x y))
≡
(match-let (((x y) (list 1 2)))
  (match-let (((a b) (list x 4)))
    (list a b x y)))
⇒
(1 4 1 2)
```

**match-letrec** ((variable expression) ...) body

Similar to **match-let**, but analogously to **letrec**, match and bind the variables in
sequence, with preceding match variables in scope.

```scheme
(match-letrec (((x y) (list 1 2)))
  ((a b) (list x 4)))
  (list a b x y))
⇒
(3 4 1 2)
```

Guile also comes with a pattern matcher specifically tailored to SXML trees, See
Section 7.16 [sxml-match], page 723.

## 7.8 Readline Support

Guile comes with an interface module to the readline library (see GNU Readline Library).
This makes interactive use much more convenient, because of the command-line editing
features of readline. Using **(ice-9 readline)**, you can navigate through the current input
line with the cursor keys, retrieve older command lines from the input history and even
search through the history entries.
7.8.1 Loading Readline Support

The module is not loaded by default and so has to be loaded and activated explicitly. This is done with two simple lines of code:

```
(use-modules (ice-9 readline))
(activate-readline)
```

The first line will load the necessary code, and the second will activate readline’s features for the REPL. If you plan to use this module often, you should save these to lines to your .guile personal startup file.

You will notice that the REPL’s behaviour changes a bit when you have loaded the readline module. For example, when you press Enter before typing in the closing parentheses of a list, you will see the *continuation* prompt, three dots: ... This gives you a nice visual feedback when trying to match parentheses. To make this even easier, *bouncing parentheses* are implemented. That means that when you type in a closing parentheses, the cursor will jump to the corresponding opening parenthesis for a short time, making it trivial to make them match.

Once the readline module is activated, all lines entered interactively will be stored in a history and can be recalled later using the cursor-up and -down keys. Readline also understands the Emacs keys for navigating through the command line and history.

When you quit your Guile session by evaluating `(quit)` or pressing Ctrl-D, the history will be saved to the file .guile_history and read in when you start Guile for the next time. Thus you can start a new Guile session and still have the (probably long-winded) definition expressions available.

You can specify a different history file by setting the environment variable GUILE_ HISTORY. And you can make Guile specific customizations to your .inputrc by testing for application ‘Guile’ (see Section “Conditional Init Constructs” in GNU Readline Library). For instance to define a key inserting a matched pair of parentheses,

```
$if Guile
   "\C-o": "()\C-b"
$endif
```

7.8.2 Readline Options

The readline interface module can be tweaked in a few ways to better suit the user’s needs. Configuration is done via the readline module’s options interface, in a similar way to the evaluator and debugging options (see Section 6.23.3 [Runtime Options], page 454).

```
readline-options
readline-enable option-name
readline-disable option-name
readline-set! option-name value
```

Accessors for the readline options. Note that unlike the enable/disable procedures, `readline-set!` is syntax, which expects an unquoted option name.

Here is the list of readline options generated by typing `(readline-options 'help)` in Guile. You can also see the default values.

```
history-file yes Use history file.
history-length 200 History length.
```
bounce-paren 500 Time (ms) to show matching opening parenthesis
(0 = off).

bracketed-paste yes Disable interpretation of control characters in pastes.

The readline options interface can only be used after loading the readline module, because it is defined in that module.

7.8.3 Readline Functions

The following functions are provided by

(use-modules (ice-9 readline))

There are two ways to use readline from Scheme code, either make calls to readline directly to get line by line input, or use the readline port below with all the usual reading functions.

readline [prompt] [Function]
Read a line of input from the user and return it as a string (without a newline at the end). prompt is the prompt to show, or the default is the string set in set-readline-prompt! below.

(readline "Type something: ") ⇒ "hello"

set-readline-input-port! port [Function]
set-readline-output-port! port [Function]
Set the input and output port the readline function should read from and write to. port must be a file port (see Section 6.14.10.1 [File Ports], page 341), and should usually be a terminal.

The default is the current-input-port and current-output-port (see Section 6.14.9 [Default Ports], page 340) when (ice-9 readline) loads, which in an interactive user session means the Unix “standard input” and “standard output”.

7.8.3.1 Readline Port

readline-port [Function]
Return a buffered input port (see Section 7.14 [Buffered Input], page 720) which calls the readline function above to get input. This port can be used with all the usual reading functions (read, read-char, etc), and the user gets the interactive editing features of readline.

There’s only a single readline port created. readline-port creates it when first called, and on subsequent calls just returns what it previously made.

activate-readline [Function]
If the current-input-port is a terminal (see Section 7.2.9 [isatty?], page 523) then enable readline for all reading from current-input-port (see Section 6.14.9 [Default Ports], page 340) and enable readline features in the interactive REPL (see Section 3.3.3 [The REPL], page 25).

(activate-readline)
(read-char)
activate-readline enables readline on current-input-port simply by a
set-current-input-port to the readline-port above. An application can do that
directly if the extra REPL features that activate-readline adds are not wanted.

set-readline-prompt! prompt1 [prompt2] [Function]
Set the prompt string to print when reading input. This is used when reading through
readline-port, and is also the default prompt for the readline function above.
prompt1 is the initial prompt shown. If a user might enter an expression across
multiple lines, then prompt2 is a different prompt to show further input required. In
the Guile REPL for instance this is an ellipsis ('...').
See set-buffered-input-continuation?! (see Section 7.14 [Buffered Input],
page 720) for an application to indicate the boundaries of logical expressions
(assuming of course an application has such a notion).

7.8.3.2 Completion

with-readline-completion-function completer thunk [Function]
Call (thunk) with completer as the readline tab completion function to be used in
any readline calls within that thunk. completer can be #f for no completion.
completer will be called as (completer text state), as described in (see Section
“How Completing Works” in GNU Readline Library). text is a partial word to be
completed, and each completer call should return a possible completion string or #f
when no more. state is #f for the first call asking about a new text then #t while
getting further completions of that text.
Here’s an example completer for user login names from the password file
(see Section 7.2.4 [User Information], page 507), much like readline’s own
rl_username_completion_function,

(define (username-completer-function text state)
  (if (not state)
      (setpwent)) ;; new, go to start of database
  (let more ((pw (getpwent)))
    (if pw
      (if (string-prefix? text (passwd:name pw))
        (passwd:name pw) ;; this name matches, return it
        (more (getpwent))) ;; doesn’t match, look at next
      (begin
        ;; end of database, close it and return #f
        (endpwent)
        #f))))

apropos-completion-function text state [Function]
A completion function offering completions for Guile functions and variables (all
defines). This is the default completion function.

filename-completion-function text state [Function]
A completion function offering filename completions. This is readline’s rl_filename_completion_function (see Section “Completion Functions” in GNU Readline Library).
make-completion-function string-list

Return a completion function which offers completions from the possibilities in string-list. Matching is case-sensitive.

7.9 Pretty Printing

The module (ice-9 pretty-print) provides the procedure pretty-print, which provides nicely formatted output of Scheme objects. This is especially useful for deeply nested or complex data structures, such as lists and vectors.

The module is loaded by entering the following:

(use-modules (ice-9 pretty-print))

This makes the procedure pretty-print available. As an example how pretty-print will format the output, see the following:

(pretty-print '(define (foo) (lambda (x)
   (cond ((zero? x) #t) ((negative? x) -x) (else
      (if (= x 1) 2 (* x x x))))))

(define (foo)
  (lambda (x)
      (cond ((zero? x) #t)
           ((negative? x) -x)
            (else (if (= x 1) 2 (* x x x))))))

pretty-print obj [port] [keyword-options]

Print the textual representation of the Scheme object obj to port. port defaults to the current output port, if not given.

The further keyword-options are keywords and parameters as follows,

#:display? flag
   If flag is true then print using display. The default is #f which means use write style. See Section 6.18.3 [Scheme Write], page 383.

#:per-line-prefix string
   Print the given string as a prefix on each line. The default is no prefix.

#:width columns
   Print within the given columns. The default is 79.

#:max-expr-width columns
   The maximum width of an expression. The default is 50.

Also exported by the (ice-9 pretty-print) module is truncated-print, a procedure to print Scheme datums, truncating the output to a certain number of characters. This is useful when you need to present an arbitrary datum to the user, but you only have one line in which to do so.

(define exp '(a b #(c d e) f . g))
(truncated-print exp #:width 10) (newline)
⊢ (a b . #)
(truncated-print exp #:width 15) (newline)
(a b # f . g)
(truncated-print exp #:width 18) (newline)
(a b #(c ...) . #)
(truncated-print exp #:width 20) (newline)
(a b #(c d e) f . g)
(truncated-print "The quick brown fox" #:width 20) (newline)
"The quick brown..."
(truncated-print (current-module) #:width 20) (newline)
#<directory (gui...>

truncated-print will not output a trailing newline. If an expression does not fit in the
given width, it will be truncated – possibly ellipsized\(^3\), or in the worst case, displayed as #.

**truncated-print** obj [port] [keyword-options] [Scheme Procedure]
Print obj, truncating the output, if necessary, to make it fit into width characters.
By default, obj will be printed using write, though that behavior can be overridden
via the display? keyword argument.
The default behaviour is to print depth-first, meaning that the entire remaining width
will be available to each sub-expression of obj – e.g., if obj is a vector, each member
of obj. One can attempt to “ration” the available width, trying to allocate it equally
to each sub-expression, via the breadth-first? keyword argument.
The further keyword-options are keywords and parameters as follows,
\[ #:display? \text{flag} \]
If flag is true then print using display. The default is #f which means use write style. see Section 6.18.3 [Scheme Write], page 383.
\[ #:width \text{columns} \]
Print within the given columns. The default is 79.
\[ #:breadth-first? \text{flag} \]
If flag is true, then allocate the available width breadth-first among elements of a compound data structure (list, vector, pair, etc.). The default is #f which means that any element is allowed to consume all of the available width.

## 7.10 Formatted Output

The **format** function is a powerful way to print numbers, strings and other objects together
with literal text under the control of a format string. This function is available from

(use-modules (ice-9 format))

A format string is generally more compact and easier than using just the standard
procedures like display, write and newline. Parameters in the output string allow various
output styles, and parameters can be taken from the arguments for runtime flexibility.

format is similar to the Common Lisp procedure of the same name, but it’s not identical
and doesn’t have quite all the features found in Common Lisp.

\(^3\) On Unicode-capable ports, the ellipsis is represented by character ‘HORIZONTAL ELLIPSIS’ (U+2026),
otherwise it is represented by three dots.
C programmers will note the similarity between `format` and `printf`, though escape sequences are marked with `~` instead of `%`, and are more powerful.

```scheme
format dest fmt arg ...  [Scheme Procedure]
Write output specified by the `fmt` string to `dest`. `dest` can be an output port, `#t` for current-output-port (see Section 6.14.9 [Default Ports], page 340), or `#f` to return the output as a string.

`fmt` can contain literal text to be output, and `~` escapes. Each escape has the form

```
~ [param [, param...] [:] [@] code
```

code is a character determining the escape sequence. The `:` and `@` characters are optional modifiers, one or both of which change the way various codes operate. Optional parameters are accepted by some codes too. Parameters have the following forms,

```
[+/-]number
An integer, with optional + or -.
'
(apostrophe)
The following character in the format string, for instance `'z` for `z`.
`v`
The next function argument as the parameter. `v` stands for “variable”, a parameter can be calculated at runtime and included in the arguments. Upper case `V` can be used too.
`#`
The number of arguments remaining. (See `~*` below for some usages.)
```

Parameters are separated by commas (,). A parameter can be left empty to keep its default value when supplying later parameters.

The following escapes are available. The code letters are not case-sensitive, upper and lower case are the same.

```
~a
~s
```

Object output. Parameters: `minwidth`, `padinc`, `minpad`, `padchar`.

`~a` outputs an argument like `display`, `~s` outputs an argument like `write` (see Section 6.18.3 [Scheme Write], page 383).

```
(format #t "~a" "foo") ⇒ foo
(format #t "~s" "foo") ⇒ "foo"
```

`~:a` and `~:s` put objects that don’t have an external representation in quotes like a string.

```
(format #t "~:a" car) ⇒ "#<primitive-procedure car>"
```

If the output is less than `minwidth` characters (default 0), it’s padded on the right with `padchar` (default space). `~@a` and `~@s` put the padding on the left instead.

```
(format #f "~5a" 'abc) ⇒ "abc "
(format #f "~5,,,'-@a" 'abc) ⇒ "--abc"
```

`minpad` is a minimum for the padding then plus a multiple of `padinc`. Ie. the padding is `minpad+N*padinc`, where `n` is the smallest integer making
the total object plus padding greater than or equal to \textit{minwidth}. The default \textit{minpad} is 0 and the default \textit{padinc} is 1 (imposing no minimum or multiple).

\begin{verbatim}
(format #f "~5,1,4a" 'abc) \Rightarrow "abc   
\end{verbatim}

$\text{~c}$

Character. Parameter: \textit{charnum}.

Output a character. The default is to simply output, as per \texttt{write-char} (see Section 6.14.11 [Venerable Port Interfaces], page 347). $\text{~@c}$ prints in \texttt{write} style. $\text{~:c}$ prints control characters (ASCII 0 to 31) in $\text{~X}$ form.

\begin{verbatim}
(format #t "~c" #\z) \Rightarrow z
(format #t "~@c" #\z) \Rightarrow #\z
(format #t "~:c" #\newline) \Rightarrow ^J
\end{verbatim}

If the \textit{charnum} parameter is given then an argument is not taken but instead the character is \texttt{(integer->char charnum)} (see Section 6.6.3 [Characters], page 129). This can be used for instance to output characters given by their ASCII code.

\begin{verbatim}
(format #t "~65c") \Rightarrow A
\end{verbatim}

$\text{~d}$

$\text{~x}$

$\text{~o}$

$\text{~b}$


Output an integer argument as a decimal, hexadecimal, octal or binary integer (respectively), in a locale-independent way.

\begin{verbatim}
(format #t "~d" 123) \Rightarrow 123
\end{verbatim}

$\text{~@d}$ etc shows a $+$ sign is shown on positive numbers.

\begin{verbatim}
(format #t "~@b" 12) \Rightarrow +1100
\end{verbatim}

If the output is less than the \textit{minwidth} parameter (default no minimum), it’s padded on the left with the \textit{padchar} parameter (default space).

\begin{verbatim}
(format #t "~5,*d" 12) \Rightarrow ***12
(format #t "~5,0d" 12) \Rightarrow 00012
(format #t "~3d" 1234) \Rightarrow 1234
\end{verbatim}

$\text{~:d}$ adds commas (or the \textit{commachar} parameter) every three digits (or the \textit{commawidth} parameter many). However, when your intent is to write numbers in a way that follows typographical conventions, using $\text{~h}$ is recommended.

\begin{verbatim}
(format #t "~:d" 1234567) \Rightarrow 1,234,567
(format #t "~10,*,'/,:d" 12345) \Rightarrow ***1/23/45
\end{verbatim}

Hexadecimal $\text{~x}$ output is in lower case, but the $\text{~( and $\text{~) case conversion directives described below can be used to get upper case.}$

\begin{verbatim}
(format #t "~x" 65261) \Rightarrow feed
(format #t "~:@(~x~)" 65261) \Rightarrow FEED
\end{verbatim}

$\text{~r}$

Integer in words, roman numerals, or a specified radix. Parameters: \textit{radix}, \textit{minwidth}, \textit{padchar}, \textit{commachar}, \textit{commawidth}.
With no parameters output is in words as a cardinal like “ten”, or \( \sim:\text{r} \) prints an ordinal like “tenth”.

\[
\begin{align*}
\text{(format #t }"\text{r}" 9) & \Rightarrow \text{nine} \quad ;; \text{cardinal} \\
\text{(format #t }"\sim:\text{r}" -9) & \Rightarrow \text{minus nine} \quad ;; \text{cardinal} \\
\text{(format #t }"\sim:\text{r}" 9) & \Rightarrow \text{ninth} \quad ;; \text{ordinal}
\end{align*}
\]

And also with no parameters, \( \sim\text{@r} \) gives roman numerals and \( \sim:\text{@r} \) gives old roman numerals. In old roman numerals there’s no “subtraction”, so 9 is \text{VIII} instead of IX. In both cases only positive numbers can be output.

\[
\begin{align*}
\text{(format #t }"\sim\text{@r}" 89) & \Rightarrow \text{LXXXIX} \quad ;; \text{roman} \\
\text{(format #t }"\sim:\text{@r}" 89) & \Rightarrow \text{LXXXVIII} \quad ;; \text{old roman}
\end{align*}
\]

When a parameter is given it means numeric output in the specified \textit{radix}. The modifiers and parameters following the radix are the same as described for \( \sim\text{d} \) etc above.

\[
\begin{align*}
\text{(format #f }"\sim:\text{3r}" 27) & \Rightarrow \text{"1000"} \quad ;; \text{base 3} \\
\text{(format #f }"\sim:\text{3,5r}" 26) & \Rightarrow \text{"222"} \quad ;; \text{base 3 width 5}
\end{align*}
\]

\(\sim f\)

Fixed-point float. Parameters: width, decimals, scale, overflowchar, padchar.

Output a number or number string in fixed-point format, ie. with a decimal point.

\[
\begin{align*}
\text{(format #t }"\sim f" 5) & \Rightarrow \text{5.0} \\
\text{(format #t }"\sim f" \"123\") & \Rightarrow \text{123.0} \\
\text{(format #t }"\sim f" \"1e-1\") & \Rightarrow \text{0.1}
\end{align*}
\]

\(\sim\text{@f}\) prints a + sign on positive numbers (including zero).

\[
\begin{align*}
\text{(format #t }"\sim\text{@f}" 0) & \Rightarrow \text{+0.0}
\end{align*}
\]

If the output is less than width characters it’s padded on the left with padchar (space by default). If the output equals or exceeds width then there’s no padding. The default for width is no padding.

\[
\begin{align*}
\text{(format #f }"\sim\text{6f}" -1.5) & \Rightarrow \text{"-1.5"} \\
\text{(format #f }"\sim\text{6,,,**f}" 23) & \Rightarrow \text{"**23.0"} \\
\text{(format #f }"\sim\text{6f}" 1234567.0) & \Rightarrow \text{"1234567.0"}
\end{align*}
\]

\textit{decimals} is how many digits to print after the decimal point, with the value rounded or padded with zeros as necessary. (The default is to output as many decimals as required.)

\[
\begin{align*}
\text{(format #t }"\sim1,2f" 3.125) & \Rightarrow \text{3.13} \\
\text{(format #t }"\sim1,2f" 1.50) & \Rightarrow \text{1.50}
\end{align*}
\]

\textit{scale} is a power of 10 applied to the value, moving the decimal point that many places. A positive \textit{scale} increases the value shown, a negative decreases it.

\[
\begin{align*}
\text{(format #t }"\sim,2f" 1234) & \Rightarrow \text{123400.0} \\
\text{(format #t }"\sim,-2f" 1234) & \Rightarrow \text{12.34}
\end{align*}
\]

If overflowchar and width are both given and if the output would exceed width, then that many overflowchars are printed instead of the value.

\[
\begin{align*}
\text{(format #t }"\sim6,,,’xf" 12345) & \Rightarrow \text{12345.}
\end{align*}
\]
=format #t "~5,,,'xf" 12345)  \(\rightarrow x\\text{xxxxx}

~h

Localized number\(^4\). Parameters: width, decimals, padchar.
Like ~f, output an exact or floating point number, but do so according to the current locale, or according to the given locale object when the : modifier is used (see Section 6.25.4 [Number Input and Output], page 462).

(=format #t "\~h" 12345.5678) ; with "C" as the current locale
\(\rightarrow 12345.5678\)

(=format #t "\~14,,,*:h" 12345.5678
  (make-locale LC_ALL "en_US"))
\(\rightarrow ***12,345.5678\)

(=format #t "\~,-2:h" 12345.5678
  (make-locale LC_NUMERIC "fr_FR"))
\(\rightarrow 12 345,56\)

~e

Exponential float. Parameters: width, mantdigits, expdigits, intdigits, overflowchar, padchar, expchar.
Output a number or number string in exponential notation.

(=format #t "\~e" 5000.25)  \(\rightarrow 5.00025E+3\)
(=format #t "\~e" "123.4")  \(\rightarrow 1.234E+2\)
(=format #f "\~e" "1e4")  \(\rightarrow 1.0E+4\)

~@e prints a + sign on positive numbers (including zero). (This is for the mantissa, a + or - sign is always shown on the exponent.)

(=format #t "\~@e" 5000.0)  \(\rightarrow +5.0E+3\)

If the output is less than width characters it's padded on the left with padchar (space by default). The default for width is to output with no padding.

(=format #f "\~10e" 1234.0)  \(\Rightarrow " 1.234E+3"\)
(=format #f "\~10,,,"*e" 0.5)  \(\Rightarrow "****5.0E-1"\)

mantdigits is the number of digits shown in the mantissa after the decimal point. The value is rounded or trailing zeros are added as necessary. The default mantdigits is to show as much as needed by the value.

(=format #f "\~,-3e" 11111.0)  \(\Rightarrow "1.111E+4"\)
(=format #f "\~,-8e" 123.0)  \(\Rightarrow "1.23000000E+2"\)

expdigits is the minimum number of digits shown for the exponent, with leading zeros added if necessary. The default for expdigits is to show only as many digits as required. At least 1 digit is always shown.

(=format #f "\~,-,1e" 1.0e99)  \(\Rightarrow "1.0E+99"\)
(=format #f "\~,-,6e" 1.0e99)  \(\Rightarrow "1.0E+00099"\)

intdigits (default 1) is the number of digits to show before the decimal point in the mantissa. intdigits can be zero, in which case the integer

\(^4\) The ~h format specifier first appeared in Guile version 2.0.6.
part is a single 0, or it can be negative, in which case leading zeros are shown after the decimal point.

(format #t "~,3e" 12345.0) ⊣ 123.45E+2
(format #t "~,0e" 12345.0)  ⊦ 0.12345E+5
(format #t "~,3e" 12345.0)  ⊦ 0.00012345E+8

If overflowchar is given then width is a hard limit. If the output would exceed width then instead that many overflowchars are printed.

(format #f "~,6,'xe" 100.0)  ⇒ "1.0E+2"
(format #f "~,3,'xe" 100.0)  ⇒ "xxx"

expchar is the exponent marker character (default E).

(format #t "~,0e" 100.0)  ⊦ 1.0e+2

~g

General float. Parameters: width, mantdigits, expdigits, intdigits, overflowchar, padchar, expchar.

Output a number or number string in either exponential format the same as ~e, or fixed-point format like ~f but aligned where the mantissa would have been and followed by padding where the exponent would have been.

Fixed-point is used when the absolute value is 0.1 or more and it takes no more space than the mantissa in exponential format, ie. basically up to mantdigits digits.

(format #f "~12,4,2g" 999.0)  ⇒ " 999.0  "
(format #f "~12,4,2g" "100000")  ⇒ " 1.0000E+05"

The parameters are interpreted as per ~e above. When fixed-point is used, the decimals parameter to ~f is established from mantdigits, so as to give a total mantdigits + 1 figures.

~$

Monetary style fixed-point float. Parameters: decimals, intdigits, width, padchar.

Output a number or number string in fixed-point format, ie. with a decimal point. decimals is the number of decimal places to show, default 2.

(format #t "~$" 5)  ⊦  5.00
(format #t "~$" "2.25")  ⊦  2.2500
(format #t "~$" "1e-2")  ⊦  0.0100

~@$

prints a + sign on positive numbers (including zero).

(format #t "~@$" 0)  ⊦ +0.00

intdigits is a minimum number of digits to show in the integer part of the value (default 1).

(format #t "~,3$" 9.5)  ⊦ 009.50
(format #t "~,0$" 0.125)  ⊦ .13

If the output is less than width characters (default 0), it’s padded on the left with padchar (default space). ~:$ puts the padding after the sign.

(format #f "~,8$" -1.5)  ⇒ " -1.50"
(format #f "~,8:$" -1.5)  ⇒ "- 1.50"
(format #f "~,8',.:@$" 3) ⇒ "+.3.00"

Note that floating point for dollar amounts is generally not a good idea, because a cent 0.01 cannot be represented exactly in the binary floating point Guile uses, which leads to slowly accumulating rounding errors. Keeping values as cents (or fractions of a cent) in integers then printing with the scale option in "f may be a better approach.

~i
Complex fixed-point float. Parameters: width, decimals, scale, overflowchar, padchar.
Output the argument as a complex number, with both real and imaginary part shown (even if one or both are zero).
The parameters and modifiers are the same as for fixed-point ~f described above. The real and imaginary parts are both output with the same given parameters and modifiers, except that for the imaginary part the @ modifier is always enabled, so as to print a + sign between the real and imaginary parts.

(format #t "~i" 1) ⊣ 1.0+0.0i

~p
Plural. No parameters.
Output nothing if the argument is 1, or 's for any other value.

(format #t "enter name~p" 1) ⊣ enter name
(format #t "enter name~p" 2) ⊣ enter names

~@p prints 'y' for 1 or 'ies' otherwise.

(format #t "pupp~@p" 1) ⊣ puppy
(format #t "pupp~@p" 2) ⊣ puppies

~:p re-uses the preceding argument instead of taking a new one, which can be convenient when printing some sort of count.

(format #t "~d cat~:p" 9) ⊣ 9 cats
(format #t "~d pupp~:@p" 5) ⊣ 5 puppies

~p is designed for English plurals and there's no attempt to support other languages. ~[ conditionals (below) may be able to help. When using gettext to translate messages ngettext is probably best though (see Section 6.25 [Internationalization], page 459).

~y
Structured printing. Parameters: width.

~y outputs an argument using pretty-print (see Section 7.9 [Pretty Printing], page 699). The result will be formatted to fit within width columns (79 by default), consuming multiple lines if necessary.

~@y outputs an argument using truncated-print (see Section 7.9 [Pretty Printing], page 699). The resulting code will be formatted to fit within width columns (79 by default), on a single line. The output will be truncated if necessary.

~:@y is like ~@y, except the width parameter is interpreted to be the maximum column to which to output. That is to say, if you are at column 10, and ~60:@y is seen, the datum will be truncated to 50 columns.
Sub-format. No parameters.
Take a format string argument and a second argument which is a list of arguments for that string, and output the result.

```
(format #t "~?" "~d ~d" '(1 2))  →  1 2
```

`~@?` takes arguments for the sub-format directly rather than in a list.

```
(format #t "~@? ~s" "~d ~d" 1 2 "foo")  →  1 2 "foo"
```

`~?` and `~k` are the same, `~k` is provided for T-Scheme compatibility.

Move forward $N$ arguments (default 1) in the argument list. `~:*` moves backwards. ($N$ cannot be negative.)

```
(format #f "~d ~2*~d" 1 2 3 4)  ⇒  "1 4"
(format #f "~d ~:*~d" 6)  ⇒  "6 6"
```

`~@*` moves to argument number $N$. The first argument is number 0 (and that's the default for $N$).

```
(format #f "~d~d again ~@*~d~d" 1 2)  ⇒  "12 again 12"
(format #f "~d~d~d again ~1@*~d~d" 1 2 3)  ⇒  "123 23"
```

A `#` move to the end followed by a `:` modifier move back can be used for an absolute position relative to the end of the argument list, a reverse of what the `@` modifier does.

```
(format #t "#align 2:*a\" 'a 'b 'c 'd\")  →  c
```

At the end of the format string the current argument position doesn't matter, any further arguments are ignored.

Advance to a column position. Parameters: `colnum`, `colinc`, `padchar`.
Output `padchar` (space by default) to move to the given `colnum` column. The start of the line is column 0, the default for `colnum` is 1.

```
(format #f "~tX")  ⇒  "X"
(format #f "~3tX")  ⇒  "X"
```

If the current column is already past `colnum`, then the move is to there plus a multiple of `colinc`, ie. column `colnum + N * colinc` for the smallest $N$ which makes that value greater than or equal to the current column. The default `colinc` is 1 (which means no further move).

```
(format #f "abcd~2,5,'.tx")  ⇒  "abcd...x"
```

`~@t` takes `colnum` as an offset from the current column. `colnum` many pad characters are output, then further padding to make the current column a multiple of `colinc`, if it isn’t already so.

```
(format #f "a~3,5'@tx")  ⇒  "a****x"
```

`~t` is implemented using `port-column` (see Section 6.14.4 [Textual I/O], page 333), so it works even there has been other output before `format`.

Tilde character. Parameter: $n$.
Output a tilde character `~`, or $n$ many if a parameter is given. Normally `~` introduces an escape sequence, `~~` is the way to output a literal tilde.
Newline. Parameter: \( n \).
Output a newline character, or \( n \) many if a parameter is given. A newline (or a few newlines) can of course be output just by including them in the format string.

Start a new line. Parameter: \( n \).
Output a newline if not already at the start of a line. With a parameter, output that many newlines, but with the first only if not already at the start of a line. So for instance 3 would be a newline if not already at the start of a line, and 2 further newlines.

Space character. Parameter: \( n \).
Output a space character, or \( n \) many if a parameter is given.
With a variable parameter this is one way to insert runtime calculated padding (~t or the various field widths can do similar things).

\[
\text{(format \#f "\textit{\~v\_foo} 4) \Rightarrow " \textit{foo}}
\]

Tab character. Parameter: \( n \).
Output a tab character, or \( n \) many if a parameter is given.

Formfeed character. Parameter: \( n \).
Output a formfeed character, or \( n \) many if a parameter is given.

Force output. No parameters.
At the end of output, call \texttt{force-output} to flush any buffers on the destination (see Section 6.14.6 [Buffering], page 336). ~! can occur anywhere in the format string, but the force is done at the end of output.
When output is to a string (destination \#f), ~! does nothing.

Newline (ie. newline character)
Continuation line. No parameters.
Skip this newline and any following whitespace in the format string, ie. don’t send it to the output. This can be used to break up a long format string for readability, but not print the extra whitespace.

\[
\text{(format \#f "abc"
   "d def"
   "d" 1 2) \Rightarrow "abc1 def2"}
\]

~:newline skips the newline but leaves any further whitespace to be printed normally.
~@newline prints the newline then skips following whitespace.

Case conversion. No parameters.
Between ~\(( and ~\)) the case of all output is changed. The modifiers on ~\(\) control the conversion.

~\( — lower case.
~@\( — upper case.


For example,

\[
\text{(format #t "~(Hello~)")} \rightarrow \text{hello}
\]
\[
\text{(format #t "~:@(Hello~)")} \rightarrow \text{HELLO}
\]

In the future it’s intended the modifiers : and @ alone will capitalize the first letters of words, as per Common Lisp format, but the current implementation of this is flawed and not recommended for use.

Case conversions do not nest, currently. This might change in the future, but if it does then it will be to Common Lisp style where the outer-most conversion has priority, overriding inner ones (making those fairly pointless).

\`{ ~} \`

Iteration. Parameter: maxreps (for \`{ ~}).

The format between \`{ and ~} is iterated. The modifiers to \`{ determine how arguments are taken. The default is a list argument with each iteration successively consuming elements from it. This is a convenient way to output a whole list.

\[
\text{(format #t "~{~d~}" '(1 2 3)) \rightarrow 123}
\]
\[
\text{(format #t "~{~s=~d ~}" "x" 1 "y" 2) \rightarrow "x"=1 "y"=2}
\]

\`{~} takes a single argument which is a list of lists, each of those contained lists gives the arguments for the iterated format.

\[
\text{(format #t "~{~d~} ~(~dx~d ~)" ((1 2) (3 4) (5 6)))}
\]
\[
\rightarrow 1x2 3x4 5x6
\]

\`{@{ ~} takes arguments directly, with each iteration successively consuming arguments.

\[
\text{(format #t "~{~d~}~" 1 2 3) \rightarrow 123}
\]
\[
\text{(format #t "~{~d~}~" "x" 1 "y" 2) \rightarrow "x"=1 "y"=2}
\]

\`{@{~} takes list arguments, one argument for each iteration, using that list for the format.

\[
\text{(format #t "~{~d~}~" ((1 2) (3 4) (5 6)))}
\]
\[
\rightarrow 1x2 3x4 5x6
\]

Iterating stops when there are no more arguments or when the maxreps parameter to \`{ is reached (default no maximum).

\[
\text{(format #t "~{~d~}~" '(1 2 3 4)) \rightarrow 12}
\]

If the format between \`{ and ~} is empty, then a format string argument is taken (before iteration argument(s)) and used instead. This allows a sub-format (like ~? above) to be iterated.

\[
\text{(format #t "~{~}~" "d" '(1 2 3)) \rightarrow 123}
\]

Iterations can be nested, an inner iteration operates in the same way as described, but of course on the arguments the outer iteration provides it. This can be used to work into nested list structures. For example in the following the inner \`{~d~}x is applied to (1 2) then (3 4 5) etc.

\[
\text{(format #t "~{~d~}~" ((1 2) (3 4 5))) \rightarrow 12x345x}
\]

See also ~^ below for escaping from iteration.
A conditional block is delimited by "[ and "], and "; separates clauses within the block. "[ takes an integer argument and that number clause is used. The first clause is number 0.

(format #f ""[peach";banana";mango"]" 1)  ⇒  "banana"
The selector parameter can be used for the clause number, instead of taking an argument.

(format #f ""2[peach";banana";mango"]")  ⇒  "mango"
If the clause number is out of range then nothing is output. Or the last clause can be ";" to use that for a number out of range.

(format #f ""[banana";mango"]" 99)  ⇒  ""
(format #f ""[banana";mango";fruit"]" 99)  ⇒  "fruit"

";[ treats the argument as a flag, and expects two clauses. The first is used if the argument is #f or the second otherwise.

(format #f ""[false";not false"]" #f)  ⇒  "false"
(format #f ""[false";not false"]" 'abc)  ⇒  "not false"

(let ((n 3))
  (format #t ""d gnu";[s are"; is"] here" n (= 1 n))
  3 gnus are here

"@[  also treats the argument as a flag, and expects one clause. If the argument is #f then no output is produced and the argument is consumed, otherwise the clause is used and the argument is not consumed, it’s left for the clause. This can be used for instance to suppress output if #f means something not available.

(format #f ""@[temperature="d"]" 27)  ⇒  "temperature=27"
(format #f ""@[temperature="d"]" #f)  ⇒  ""

""  Escape. Parameters: val1, val2, val3.
Stop formatting if there are no more arguments. This can be used for instance to have a format string adapt to a variable number of arguments.

(format #t ""d"; d" 1)  ⇒  1
(format #t ""d"; d" 1 2)  ⇒  1 2
Within a "{ ~} iteration, "" stops the current iteration step if there are no more arguments to that step, but continuing with possible further steps and the rest of the format. This can be used for instance to avoid a separator on the last iteration, or to adapt to variable length argument lists.

(format #f ""{d"; d"} go" '(1 2 3))  ⇒  "1/2/3 go"
(format #f "";{ ~d"; d"} go" '((1) (2 3)))  ⇒  "1 23 go"
Within a ~? sub-format, "" operates just on that sub-format. If it terminates the sub-format then the originating format will still continue.

(format #t ""? items" ""d"; d" '(1))  ⇒  1 items
(format #t ""? items" ""d"; d" '(1 2))  ⇒  1 2 items
The parameters to "~" (which are numbers) change the condition used to terminate. For a single parameter, termination is when that value is zero (notice this makes plain "~" equivalent to "~#"). For two parameters, termination is when those two are equal. For three parameters, termination is when \( \text{val1} \leq \text{val2} \) and \( \text{val2} \leq \text{val3} \).

"~q" Inquiry message. Insert a copyright message into the output.

"~:q" inserts the format implementation version.

It’s an error if there are not enough arguments for the escapes in the format string, but any excess arguments are ignored.

Iterations "~{~}" and conditionals "~[~;~]" can be nested, but must be properly nested, meaning the inner form must be entirely within the outer form. So it’s not possible, for instance, to try to conditionallyize the endpoint of an iteration.

\[
\begin{align*}
\text{(format } & \text{#t } "\text{~{ ~[ ~\ldots ~\} ~}}\ldots") ;; \text{good} \\
\text{(format } & \text{#t } "\text{~{ ~[ ~\ldots ~\} ~\ldots ~\] ~}}\ldots") ;; \text{bad}
\end{align*}
\]

The same applies to case conversions "(~), they must properly nest with respect to iterations and conditionals (though currently a case conversion cannot nest within another case conversion).

When a sub-format (~?) is used, that sub-format string must be self-contained. It cannot for instance give a "~{ to begin an iteration form and have the ~} up in the originating format, or similar.

Guile contains a format procedure even when the module (ice-9 format) is not loaded. The default format is simple-format (see Section 6.14.5 [Simple Output], page 335), it doesn’t support all escape sequences documented in this section, and will signal an error if you try to use one of them. The reason for two versions is that the full format is fairly large and requires some time to load. simple-format is often adequate too.

7.11 File Tree Walk

The functions in this section traverse a tree of files and directories. They come in two flavors: the first one is a high-level functional interface, and the second one is similar to the C ftw and nftw routines (see Section “Working with Directory Trees” in GNU C Library Reference Manual).

(use-modules (ice-9 ftw))

file-system-tree file-name [enter? [stat]]

[Scheme Procedure]

Return a tree of the form (file-name stat children ...) where stat is the result of (stat file-name) and children are similar structures for each file contained in file-name when it designates a directory.

The optional enter? predicate is invoked as (enter? name stat) and should return true to allow recursion into directory name; the default value is a procedure that always returns #t. When a directory does not match enter?, it nonetheless appears in the resulting tree, only with zero children.
The stat argument is optional and defaults to lstat, as for file-system-fold (see below.)

The example below shows how to obtain a hierarchical listing of the files under the module/language directory in the Guile source tree, discarding their stat info:

```scheme
(use-modules (ice-9 match))

(define remove-stat
  ;; Remove the 'stat' object the 'file-system-tree' provides
  ;; for each file in the tree.
  (match-lambda
    ((name stat) ; flat file
      name)
    ((name stat children ...) ; directory
      (list name (map remove-stat children)))))

(let ((dir (string-append (assq-ref %guile-build-info 'top_srcdir)
                    
       "./module/language")))
  (remove-stat (file-system-tree dir)))
⇒
("language"
  ("value" ("spec.go" "spec.scm"))
  ("scheme"
   ("spec.go"
    "spec.scm"
    "compile-tree-il.scm"
    "decompile-tree-il.scm"
    "decompile-tree-il.go"
    "compile-tree-il.go"))
  ("tree-il"
   ("spec.go"
    "fix-letrec.go"
    "inline.go"
    "fix-letrec.scm"
    "compile-gilil.go"
    "spec.scm"
    "optimize.scm"
    "primitives.scm"
    ...
    )))

It is often desirable to process directories entries directly, rather than building up a tree of entries in memory, like file-system-tree does. The following procedure, a combinator, is designed to allow directory entries to be processed directly as a directory tree is traversed; in fact, file-system-tree is implemented in terms of it.
file-system-fold  enter?  leaf  down  up  skip  error  init

file-name [stat]

[Scheme Procedure]

Traverse the directory at file-name, recursively, and return the result of the successive applications of the leaf, down, up, and skip procedures as described below.

Enter sub-directories only when (enter? path stat result) returns true. When a sub-directory is entered, call (down path stat result), where path is the path of the sub-directory and stat the result of (false-if-exception (stat path)); when it is left, call (up path stat result).

For each file in a directory, call (leaf path stat result).

When enter? returns #f, or when an unreadable directory is encountered, call (skip path stat result).

When file-name names a flat file, (leaf path stat init) is returned.

When an opendir or stat call fails, call (error path stat errno result), with errno being the operating system error number that was raised—e.g., EACCES—and stat either #f or the result of the stat call for that entry, when available.

The special . and .. entries are not passed to these procedures. The path argument to the procedures is a full file name—e.g., "../foo/bar/gnu"; if file-name is an absolute file name, then path is also an absolute file name. Files and directories, as identified by their device/inode number pair, are traversed only once.

The optional stat argument defaults to lstat, which means that symbolic links are not followed; the stat procedure can be used instead when symbolic links are to be followed (see Section 7.2.3 [File System], page 500).

The example below illustrates the use of file-system-fold:

(define (total-file-size file-name)
  "Return the size in bytes of the files under FILE-NAME (similar to 'du --apparent-size' with GNU Coreutils.)"

  (define (enter? name stat result)
    ;; Skip version control directories.
    (not (member (basename name) '(".git" ".svn" "CVS"))))

  (define (leaf name stat result)
    ;; Return RESULT plus the size of the file at NAME.
    (+ result (stat:size stat)))

  (define (down name stat result) result)
  (define (up name stat result) result)
  (define (skip name stat result) result)
  (define (error name stat errno result)
    (format (current-error-port) "warning: ~a: ~a~%" name (strerror errno)))

  (define (total-file-size file-name)
    "Return the size in bytes of the files under FILE-NAME (similar to 'du --apparent-size' with GNU Coreutils.)"

    (define (enter? name stat result)
      ;; Skip version control directories.
      (not (member (basename name) '(".git" ".svn" "CVS"))))

    (define (leaf name stat result)
      ;; Return RESULT plus the size of the file at NAME.
      (+ result (stat:size stat)))

    (define (down name stat result) result)
    (define (up name stat result) result)
    (define (skip name stat result) result)
    (define (error name stat errno result)
      (format (current-error-port) "warning: ~a: ~a~%" name (strerror errno)))

    (define (total-file-size file-name)  
      "Return the size in bytes of the files under FILE-NAME (similar to 'du --apparent-size' with GNU Coreutils.)"

      (define (enter? name stat result)
        ;; Skip version control directories.
        (not (member (basename name) '(".git" ".svn" "CVS"))))

        (define (leaf name stat result)
          ;; Return RESULT plus the size of the file at NAME.
          (+ result (stat:size stat)))

          (define (down name stat result) result)
          (define (up name stat result) result)
          (define (skip name stat result) result)
          (define (error name stat errno result)
            (format (current-error-port) "warning: ~a: ~a~%" name (strerror errno)))

          (define (total-file-size file-name)
            "Return the size in bytes of the files under FILE-NAME (similar to 'du --apparent-size' with GNU Coreutils.)"

            (define (enter? name stat result)
              ;; Skip version control directories.
              (not (member (basename name) '(".git" ".svn" "CVS"))))

              (define (leaf name stat result)
                ;; Return RESULT plus the size of the file at NAME.
                (+ result (stat:size stat)))

                (define (down name stat result) result)
                (define (up name stat result) result)
                (define (skip name stat result) result)
                (define (error name stat errno result)
                  (format (current-error-port) "warning: ~a: ~a~%" name (strerror errno)))
result)

(file-system-fold enter? leaf down up skip error
  0 ; initial counter is zero bytes
  file-name))

(total-file-size ".")
⇒ 8217554

(total-file-size "/dev/null")
⇒ 0

The alternative C-like functions are described below.

**scandir** name [select? [entry<?]] [Scheme Procedure]

Return the list of the names of files contained in directory name that match predicate select? (by default, all files). The returned list of file names is sorted according to entry<?, which defaults to string-locale<? such that file names are sorted in the locale's alphabetical order (see Section 6.25.2 [Text Collation], page 460). Return #f when name is unreadable or is not a directory.

This procedure is modeled after the C library function of the same name (see Section “Scanning Directory Content” in GNU C Library Reference Manual).

**ftw** startname proc ['hash-size n] [Scheme Procedure]

Walk the file system tree descending from startname, calling proc for each file and directory.

Hard links and symbolic links are followed. A file or directory is reported to proc only once, and skipped if seen again in another place. One consequence of this is that ftw is safe against circularly linked directory structures.

Each proc call is (proc filename statinfo flag) and it should return #t to continue, or any other value to stop.

filename is the item visited, being startname plus a further path and the name of the item. statinfo is the return from stat (see Section 7.2.3 [File System], page 500) on filename. flag is one of the following symbols,

- **regular** filename is a file, this includes special files like devices, named pipes, etc.
- **directory** filename is a directory.
- **invalid-stat** An error occurred when calling stat, so nothing is known. statinfo is #f in this case.
- **directory-not-readable** filename is a directory, but one which cannot be read and hence won’t be recursed into.
- **symlink** filename is a dangling symbolic link. Symbolic links are normally followed and their target reported, the link itself is reported if the target does not exist.
The return value from `ftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from `proc` which caused the stop.

Optional argument symbol `hash-size` and an integer can be given to set the size of the hash table used to track items already visited. (see Section 6.6.22.2 [Hash Table Reference], page 239)

In the current implementation, returning non-`#t` from `proc` is the only valid way to terminate `ftw`. `proc` must not use `throw` or similar to escape.

```
nftw startname proc ['chdir] ['depth] ['hash-size n] ['mount]   [Scheme Procedure]

Walk the file system tree starting at `startname`, calling `proc` for each file and directory. `nftw` has extra features over the basic `ftw` described above.

Like `ftw`, hard links and symbolic links are followed. A file or directory is reported to `proc` only once, and skipped if seen again in another place. One consequence of this is that `nftw` is safe against circular linked directory structures.

Each `proc` call is `(proc filename statinfo flag base level)` and it should return `#t` to continue, or any other value to stop.

`filename` is the item visited, being `startname` plus a further path and the name of the item. `statinfo` is the return from `stat` on `filename` (see Section 7.2.3 [File System], page 500). `base` is an integer offset into `filename` which is where the basename for this item begins. `level` is an integer giving the directory nesting level, starting from 0 for the contents of `startname` (or that item itself if it’s a file). `flag` is one of the following symbols,

- `regular` `filename` is a file, including special files like devices, named pipes, etc.
- `directory` `filename` is a directory.
- `directory-processed` `filename` is a directory, and its contents have all been visited. This flag is given instead of `directory` when the `depth` option below is used.
- `invalid-stat` An error occurred when applying `stat` to `filename`, so nothing is known about it. `statinfo` is `#f` in this case.
- `directory-not-readable` `filename` is a directory, but one which cannot be read and hence won’t be recursed into.
- `stale-symlink` `filename` is a dangling symbolic link. Links are normally followed and their target reported, the link itself is reported if its target does not exist.
- `symlink` When the `physical` option described below is used, this indicates `filename` is a symbolic link whose target exists (and is not being followed).
The following optional arguments can be given to modify the way nftw works. Each is passed as a symbol (and hash-size takes a following integer value).

- **chdir** Change to the directory containing the item before calling proc. When nftw returns the original current directory is restored.

  Under this option, generally the base parameter to each proc call should be used to pick out the base part of the filename. The filename is still a path but with a changed directory it won’t be valid (unless the startname directory was absolute).

- **depth** Visit files “depth first”, meaning proc is called for the contents of each directory before it’s called for the directory itself. Normally a directory is reported first, then its contents.

  Under this option, the flag to proc for a directory is directory-processed instead of directory.

- **hash-size n** Set the size of the hash table used to track items already visited. (see Section 6.6.22.2 [Hash Table Reference], page 239)

- **mount** Don’t cross a mount point, meaning only visit items on the same file system as startname (ie. the same stat:dev).

- **physical** Don’t follow symbolic links, instead report them to proc as symlink. Dangling links (those whose target doesn’t exist) are still reported as stale-symlink.

The return value from nftw is #t if it ran to completion, or otherwise the non-#t value from proc which caused the stop.

In the current implementation, returning non-#t from proc is the only valid way to terminate ftw. proc must not use throw or similar to escape.

### 7.12 Queues

The functions in this section are provided by

```
(use-modules (ice-9 q))
```

This module implements queues holding arbitrary scheme objects and designed for efficient first-in / first-out operations.

- **make-q** creates a queue, and objects are entered and removed with enq! and deq!.

- **q-push!** and **q-pop!** can be used too, treating the front of the queue like a stack.

- **make-q**
  
  Return a new queue.

- **q? obj**
  
  Return #t if obj is a queue, or #f if not.

  Note that queues are not a distinct class of objects but are implemented with cons cells. For that reason certain list structures can get #t from q?.
enq!  q  obj
Add obj to the rear of q, and return q.

deq!  q
q-pop!  q
Remove and return the front element from q. If q is empty, a q-empty exception is thrown.

deq! and q-pop! are the same operation, the two names just let an application match enq! with deq!, or q-push! with q-pop!.

q-push!  q  obj
Add obj to the front of q, and return q.

q-length q
Return the number of elements in q.

q-empty?  q
Return true if q is empty.

q-empty-check  q
Throw a q-empty exception if q is empty.

q-front q
Return the first element of q (without removing it). If q is empty, a q-empty exception is thrown.

q-rear q
Return the last element of q (without removing it). If q is empty, a q-empty exception is thrown.

q-remove!  q  obj
Remove all occurrences of obj from q, and return q. obj is compared to queue elements using eq?.

The q-empty exceptions described above are thrown just as (throw 'q-empty), there’s no message etc like an error throw.

A queue is implemented as a cons cell, the car containing a list of queued elements, and the cdr being the last cell in that list (for ease of enqueuing).

(list . last-cell)
If the queue is empty, list is the empty list and last-cell is #f.

An application can directly access the queue list if desired, for instance to search the elements or to insert at a specific point.

sync-q!  q
Recompute the last-cell field in q.

All the operations above maintain last-cell as described, so normally there’s no need for sync-q!. But if an application modifies the queue list then it must either maintain last-cell similarly, or call sync-q! to recompute it.
7.13 Streams

This section documents Guile’s legacy stream module. For a more complete and portable stream library, see Section 7.5.28 [SRFI-41], page 625.

A stream represents a sequence of values, each of which is calculated only when required. This allows large or even infinite sequences to be represented and manipulated with familiar operations like “car”, “cdr”, “map” or “fold”. In such manipulations only as much as needed is actually held in memory at any one time. The functions in this section are available from

```
(use-modules (ice-9 streams))
```

Streams are implemented using promises (see Section 6.18.9 [Delayed Evaluation], page 393), which is how the underlying calculation of values is made only when needed, and the values then retained so the calculation is not repeated.

Here is a simple example producing a stream of all odd numbers,

```
(define odds (make-stream (lambda (state)
                        (cons state (+ state 2)))
                      1))
```

```
(stream-car odds) ⇒ 1
(stream-car (stream-cdr odds)) ⇒ 3
```

`stream-map` could be used to derive a stream of odd squares,

```
(define (square n) (* n n))
(define oddsquares (stream-map square odds))
```

These are infinite sequences, so it’s not possible to convert them to a list, but they could be printed (infinitely) with for example

```
(stream-for-each (lambda (n sq)
                (format #t "~a squared is ~a\n" n sq))
               odds oddsquares)
```

```
-1
1 squared is 1
3 squared is 9
5 squared is 25
7 squared is 49
...
```

### make-stream proc initial-state

[Scheme Procedure]

Return a new stream, formed by calling `proc` successively.

Each call is `(proc state)`, it should return a pair, the car being the value for the stream, and the cdr being the new state for the next call. For the first call state is the given initial-state. At the end of the stream, proc should return some non-pair object.

### stream-car stream

[Scheme Procedure]

Return the first element from `stream`. stream must not be empty.

### stream-cdr stream

[Scheme Procedure]

Return a stream which is the second and subsequent elements of `stream`. stream must not be empty.
stream-null? stream                      [Scheme Procedure]
       Return true if stream is empty.

list->stream list                         [Scheme Procedure]
vector->stream vector                    [Scheme Procedure]
       Return a stream with the contents of list or vector.

list or vector should not be modified subsequently, since it’s unspecified whether
changes there will be reflected in the stream returned.

port->stream port readproc               [Scheme Procedure]
       Return a stream which is the values obtained by reading from port using readproc.
       Each read call is (readproc port), and it should return an EOF object (see
Section 6.14.2 [Binary I/O], page 330) at the end of input.

For example a stream of characters from a file,
   (port->stream (open-input-file "/foo/bar.txt") read-char)

stream->list stream                      [Scheme Procedure]
       Return a list which is the entire contents of stream.

stream->reversed-list stream             [Scheme Procedure]
       Return a list which is the entire contents of stream, but in reverse order.

stream->list&length stream               [Scheme Procedure]
       Return two values (see Section 6.13.7 [Multiple Values], page 309), being firstly a list
which is the entire contents of stream, and secondly the number of elements in that
list.

stream->reversed-list&length stream      [Scheme Procedure]
       Return two values (see Section 6.13.7 [Multiple Values], page 309) being firstly a list
which is the entire contents of stream, but in reverse order, and secondly the number
of elements in that list.

stream->vector stream                    [Scheme Procedure]
       Return a vector which is the entire contents of stream.

stream-fold proc init stream1 stream2 ...
       [Function]
       Apply proc successively over the elements of the given streams, from first to last until
the end of the shortest stream is reached. Return the result from the last proc call.

Each call is (proc elem1 elem2 ... prev), where each elem is from the corresponding
stream. prev is the return from the previous proc call, or the given init for the first
call.

stream-for-each proc stream1 stream2 ...
       [Function]
       Call proc on the elements from the given streams. The return value is unspecified.

Each call is (proc elem1 elem2 ...), where each elem is from the corresponding
stream. stream-for-each stops when it reaches the end of the shortest stream.
**stream-map** proc stream1 stream2 ...  

[Function]

Return a new stream which is the results of applying proc to the elements of the given streams.

Each call is (proc elem1 elem2 ...), where each elem is from the corresponding stream. The new stream ends when the end of the shortest given stream is reached.

### 7.14 Buffered Input

The following functions are provided by

```
(use-modules (ice-9 buffered-input))
```

A buffered input port allows a reader function to return chunks of characters which are to be handed out on reading the port. A notion of further input for an application level logical expression is maintained too, and passed through to the reader.

**make-buffered-input-port** reader

[Scheme Procedure]

Create an input port which returns characters obtained from the given reader function. reader is called (reader cont), and should return a string or an EOF object.

The new port gives precisely the characters returned by reader, nothing is added, so if any newline characters or other separators are desired they must come from the reader function.

The cont parameter to reader is #f for initial input, or #t when continuing an expression. This is an application level notion, set with set-buffered-input-continuation?! below. If the user has entered a partial expression then it allows reader for instance to give a different prompt to show more is required.

**make-line-buffered-input-port** reader

[Scheme Procedure]

Create an input port which returns characters obtained from the specified reader function, similar to make-buffered-input-port above, but where reader is expected to be a line-oriented.

reader is called (reader cont), and should return a string or an EOF object as above. Each string is a line of input without a newline character, the port code inserts a newline after each string.

**set-buffered-input-continuation?!** port cont

[Scheme Procedure]

Set the input continuation flag for a given buffered input port.

An application uses this by calling with a cont flag of #f when beginning to read a new logical expression. For example with the Scheme read function (see Section 6.18.2 [Scheme Read], page 382),

```
(define my-port (make-buffered-input-port my-reader))

(set-buffered-input-continuation?! my-port #f)
(let ((obj (read my-port)))
  ...
```


7.15 Expect

The macros in this section are made available with:

```
(use-modules (ice-9 expect))
```

`expect` is a macro for selecting actions based on the output from a port. The name comes from a tool of similar functionality by Don Libes. Actions can be taken when a particular string is matched, when a timeout occurs, or when end-of-file is seen on the port. The `expect` macro is described below; `expect-strings` is a front-end to `expect` based on `regexec` (see the regular expression documentation).

`expect-strings` clause ...

By default, `expect-strings` will read from the current input port. The first term in each clause consists of an expression evaluating to a string pattern (regular expression). As characters are read one-by-one from the port, they are accumulated in a buffer string which is matched against each of the patterns. When a pattern matches, the remaining expression(s) in the clause are evaluated and the value of the last is returned. For example:

```
(with-input-from-file "/etc/passwd"
  (lambda ()
    (expect-strings
      ("nobody" (display "Got a nobody user.\n")
       (display "That's no problem.\n"))
      ("daemon" (display "Got a daemon user.\n"))))
```

The regular expression is compiled with the REG_NEWLINE flag, so that the `^` and `\$` anchors will match at any newline, not just at the start and end of the string.

There are two other ways to write a clause:

- The expression(s) to evaluate can be omitted, in which case the result of the regular expression match (converted to strings, as obtained from `regexec` with match-pick set to `""`) will be returned if the pattern matches.
- The symbol `=>` can be used to indicate that the expression is a procedure which will accept the result of a successful regular expression match. E.g.,

```
("daemon" => write)
("\d(aemon)" => (lambda args (for-each write args)))
("\d(\arm)on" => (lambda (all sub)
    (write all) (newline)
    (write sub) (newline)))
```

The order of the substrings corresponds to the order in which the opening brackets occur.

A number of variables can be used to control the behaviour of `expect` (and `expect-strings`). Most have default top-level bindings to the value `#f`, which produces the default behaviour. They can be redefined at the top level or locally bound in a form enclosing the `expect` expression.

`expect-port`

A port to read characters from, instead of the current input port.
**expect-timeout**

*expect* will terminate after this number of seconds, returning `#f` or the value returned by *expect-timeout-proc*.

**expect-timeout-proc**

A procedure called if timeout occurs. The procedure takes a single argument: the accumulated string.

**expect-eof-proc**

A procedure called if end-of-file is detected on the input port. The procedure takes a single argument: the accumulated string.

**expect-char-proc**

A procedure to be called every time a character is read from the port. The procedure takes a single argument: the character which was read.

**expect-strings-compile-flags**

Flags to be used when compiling a regular expression, which are passed to `make-regexp` See Section 6.15.1 [Regexp Functions], page 356. The default value is `regexp/newline`.

**expect-strings-exec-flags**

Flags to be used when executing a regular expression, which are passed to `regexp-exec` See Section 6.15.1 [Regexp Functions], page 356. The default value is `regexp/noteol`, which prevents `\$` from matching the end of the string while it is still accumulating, but still allows it to match after a line break or at the end of file.

Here's an example using all of the variables:

```scheme
(let ((expect-port (open-input-file "/etc/passwd")))
  (expect-timeout 1)
  (expect-timeout-proc
   (lambda (s) (display "Times up!\n"))))
  (expect-eof-proc
   (lambda (s) (display "Reached the end of the file!\n"))))
  (expect-char-proc display)
  (expect-strings-compile-flags (logior regexp/newline regexp/icase))
  (expect-strings-exec-flags 0))
  (expect-strings
   ("nobody" (display "Got a nobody user\n"))))
```

**expect clause . . .**

*expect* is used in the same way as *expect-strings*, but tests are specified not as patterns, but as procedures. The procedures are called in turn after each character is read from the port, with two arguments: the value of the accumulated string and a flag to indicate whether end-of-file has been reached. The flag will usually be `#f`, but if end-of-file is reached, the procedures are called an additional time with the final accumulated string and `#t`.

The test is successful if the procedure returns a non-false value.

If the `=>` syntax is used, then if the test succeeds it must return a list containing the arguments to be provided to the corresponding expression.
In the following example, a string will only be matched at the beginning of the file:

```scheme
(let ((expect-port (open-input-file "/etc/passwd")))
  (expect
   ((lambda (s eof?) (string=? s "fnord!"))
    (display "Got a nobody user!\n"))))
```

The control variables described for `expect-strings` also influence the behaviour of `expect`, with the exception of variables whose names begin with `expect-strings-`.

### 7.16 sxml-match: Pattern Matching of SXML

The `(sxml match)` module provides syntactic forms for pattern matching of SXML trees, in a “by example” style reminiscent of the pattern matching of the `syntax-rules` and `syntax-case` macro systems. See Section 7.20 [SXML], page 733, for more information on SXML.

The following example

5 This example is taken from a paper by Krishnamurthi et al. Their paper was the first to show the usefulness of the `syntax-rules` style of pattern matching for transformation of XML, though the language described, XT3D, is an XML language.

provides a brief illustration, transforming a music album catalog language into HTML.

```scheme
(define (album->html x)
  (sxml-match x
    [(album (@ (title ,t)) (catalog (num ,n) (fmt ,f)) ...) 
     (ul (li ,t)
         (li (b ,n) (i ,f)) ...)])
)
```

Three macros are provided: `sxml-match`, `sxml-match-let`, and `sxml-match-let*`.

Compared to a standard s-expression pattern matcher (see Section 7.7 [Pattern Matching], page 691), `sxml-match` provides the following benefits:

- matching of SXML elements does not depend on any degree of normalization of the SXML;
- matching of SXML attributes (within an element) is under-ordered; the order of the attributes specified within the pattern need not match the ordering with the element being matched;
- all attributes specified in the pattern must be present in the element being matched; in the spirit that XML is ’extensible’, the element being matched may include additional attributes not specified in the pattern.

The present module is a descendant of WebIt!, and was inspired by an s-expression pattern matcher developed by Erik Hilsdale, Dan Friedman, and Kent Dybvig at Indiana University.

### Syntax

`sxml-match` provides `case`-like form for pattern matching of XML nodes.

```
sxml-match input-expression clause1 clause2 ... [Scheme Syntax]
```

Match `input-expression`, an SXML tree, according to the given `clauses` (one or more), each consisting of a pattern and one or more expressions to be evaluated if the pattern
match succeeds. Optionally, each clause within \texttt{sxml-match} may include a guard expression.

The pattern notation is based on that of Scheme’s \texttt{syntax-rules} and \texttt{syntax-case} macro systems. The grammar for the \texttt{sxml-match} syntax is given below:

\begin{verbatim}
match-form ::= (sxml-match input-expression
               clause+)
clause ::= [node-pattern action-expression+]
          | [node-pattern (guard expression*) action-expression+]

node-pattern ::= literal-pattern
              | pat-var-or-cata
              | element-pattern
              | list-pattern

literal-pattern ::= string
                | character
                | number
                | #t
                | #f

attr-list-pattern ::= (@ attribute-pattern*)
                    | (@ attribute-pattern* . pat-var-or-cata)

attribute-pattern ::= (tag-symbol attr-val-pattern)

attr-val-pattern ::= literal-pattern
                  | pat-var-or-cata
                  | (pat-var-or-cata default-value-expr)

element-pattern ::= (tag-symbol attr-list-pattern?)
                  | (tag-symbol attr-list-pattern? nodeset-pattern)
                  | (tag-symbol attr-list-pattern? nodeset-pattern? . pat-var-or-cata)

list-pattern ::= (list nodeset-pattern)
              | (list nodeset-pattern? . pat-var-or-cata)
              | (list)

nodeset-pattern ::= node-pattern
                  | node-pattern ...
                  | node-pattern nodeset-pattern
                  | node-pattern ... nodeset-pattern

pat-var-or-cata ::= (unquote var-symbol)
                   | (unquote [var-symbol*])
\end{verbatim}
Within a list or element body pattern, ellipses may appear only once, but may be followed by zero or more node patterns.

Guard expressions cannot refer to the return values of catamorphisms.

Ellipses in the output expressions must appear only in an expression context; ellipses are not allowed in a syntactic form.

The sections below illustrate specific aspects of the \texttt{sxml-match} pattern matcher.

### Matching XML Elements

The example below illustrates the pattern matching of an XML element:

\begin{verbatim}
(sxml-match '(e (@ (i 1)) 3 4 5)
  [(e (@ (i ,d)) ,a ,b ,c) (list d a b c)]
  [,otherwise #f])
\end{verbatim}

Each clause in \texttt{sxml-match} contains two parts: a pattern and one or more expressions which are evaluated if the pattern is successfully match. The example above matches an element \texttt{e} with an attribute \texttt{i} and three children.

Pattern variables must be “unquoted” in the pattern. The above expression binds \texttt{d} to \texttt{1}, \texttt{a} to \texttt{3}, \texttt{b} to \texttt{4}, and \texttt{c} to \texttt{5}.

### Ellipses in Patterns

As in \texttt{syntax-rules}, ellipses may be used to specify a repeated pattern. Note that the pattern \texttt{item ...} specifies zero-or-more matches of the pattern \texttt{item}.

The use of ellipses in a pattern is illustrated in the code fragment below, where nested ellipses are used to match the children of repeated instances of an \texttt{a} element, within an element \texttt{d}.

\begin{verbatim}
(define x '(d (a 1 2 3) (a 4 5) (a 6 7 8) (a 9 10)))

(sxml-match x
  [(d (a ,b ...) ...) (list (list b ...) ...)])
\end{verbatim}

The above expression returns a value of \texttt{((1 2 3) (4 5) (6 7 8) (9 10))}.

### Ellipses in Quasiquote'd Output

Within the body of an \texttt{sxml-match} form, a slightly extended version of quasiquote is provided, which allows the use of ellipses. This is illustrated in the example below.

\begin{verbatim}
(sxml-match '(e 3 4 5 6 7)
  [(e ,i ... 6 7) ('"start" ,(list 'wrap i) ... "end")]
  [,otherwise #f])
\end{verbatim}

The general pattern is that \texttt{'(something ,i ...)} is rewritten as \texttt{'(something ,@i)}.

### Matching Nodesets

A nodeset pattern is designated by a list in the pattern, beginning the identifier list. The example below illustrates matching a nodeset.

\begin{verbatim}
(sxml-match '("i" "j" "k" "l" "m")
\end{verbatim}
This example wraps each nodeset item in an HTML paragraph element. This example can be rewritten and simplified through using ellipsis:

```scheme
(sxml-match '("i" "j" "k" "l" "m")
  [(list ,i ...) '((p ,i) ...)])
```

This version will match nodesets of any length, and wrap each item in the nodeset in an HTML paragraph element.

**Matching the “Rest” of a Nodeset**

Matching the “rest” of a nodeset is achieved by using a . rest) pattern at the end of an element or nodeset pattern.

This is illustrated in the example below:

```scheme
(sxml-match '(e 3 (f 4 5 6) 7)
  [(e ,a (f . ,y) ,d)
   (list a y d)])
```

The above expression returns (3 (4 5 6) 7).

**Matching the Unmatched Attributes**

Sometimes it is useful to bind a list of attributes present in the element being matched, but which do not appear in the pattern. This is achieved by using a . rest) pattern at the end of the attribute list pattern. This is illustrated in the example below:

```scheme
(sxml-match '(a (@ (z 1) (y 2) (x 3)) 4 5 6)
  [(a (@ (y ,www) . ,qqq) ,t ,u ,v)
   (list www qqq t u v)])
```

The above expression matches the attribute y and binds a list of the remaining attributes to the variable qqq. The result of the above expression is (2 ((z 1) (x 3)) 4 5 6).

This type of pattern also allows the binding of all attributes:

```scheme
(sxml-match '(a (@ (z 1) (y 2) (x 3)))
  [(a (@ . ,qqq))
   qqq])
```

**Default Values in Attribute Patterns**

It is possible to specify a default value for an attribute which is used if the attribute is not present in the element being matched. This is illustrated in the following example:

```scheme
(sxml-match '(e 3 4 5)
  [(e (@ (z (,d 1))) ,a ,b ,c) (list d a b c)])
```

The value 1 is used when the attribute z is absent from the element e.

**Guards in Patterns**

Guards may be added to a pattern clause via the guard keyword. A guard expression may include zero or more expressions which are evaluated only if the pattern is matched. The body of the clause is only evaluated if the guard expressions evaluate to #t.
The use of guard expressions is illustrated below:

```scheme
(sxml-match '(a 2 3)
  ((a ,n) (guard (number? n)) n)
  ((a ,m ,n) (guard (number? m) (number? n)) (+ m n)))
```

**Catamorphisms**

The example below illustrates the use of explicit recursion within an `sxml-match` form. This example implements a simple calculator for the basic arithmetic operations, which are represented by the XML elements `plus`, `minus`, `times`, and `div`.

```scheme
(define simple-eval
  (lambda (x)
    (sxml-match x
      [,i (guard (integer? i)) i]
      [(plus ,x ,y) (+ (simple-eval x) (simple-eval y))]
      [(times ,x ,y) (* (simple-eval x) (simple-eval y))]
      [(minus ,x ,y) (- (simple-eval x) (simple-eval y))]
      [(div ,x ,y) (/ (simple-eval x) (simple-eval y))]
      [,otherwise (error "simple-eval: invalid expression" x)])
  )))
```

Using the catamorphism feature of `sxml-match`, a more concise version of `simple-eval` can be written. The pattern `[,]` recursively invokes the pattern matcher on the value bound in this position.

```scheme
(define simple-eval
  (lambda (x)
    (sxml-match x
      [,i (guard (integer? i)) i]
      [(plus ,[x] ,[y]) (+ x y)]
      [(times ,[x] ,[y]) (* x y)]
      [(minus ,[x] ,[y]) (- x y)]
      [(div ,[x] ,[y]) (/ x y)]
      [,otherwise (error "simple-eval: invalid expression" x)])
  )))
```

**Named-Catamorphisms**

It is also possible to explicitly name the operator in the “cata” position. Where `[id*]` recurs to the top of the current `sxml-match`, `[cata -> id*]` recurs to `cata`. `cata` must evaluate to a procedure which takes one argument, and returns as many values as there are identifiers following `->`.

Named catamorphism patterns allow processing to be split into multiple, mutually recursive procedures. This is illustrated in the example below: a transformation that formats a “TV Guide” into HTML.

```scheme
(define (tv-guide->html g)
  (define (cast-list cl)
    (sxml-match cl
      [(CastList (CastMember (Character (Name ,ch)) (Actor (Name ,a))) ...) '((div (ul (li ,ch "": " ,a) ...)))])
  )

(define (prog p)
(sxml-match p
  '[(Program (Start ,start-time) (Duration ,dur) (Series ,series-title) (Description ,desc ...))]

(sxml-match g
  '[(TVGuide (@ (start ,start-date) (end ,end-date)) (Channel (Name ,nm) ,[prog -> p] ...) ...)]

(sxml-match-let ([(a ,i ,j) '(a 1 2)])
  (+ i j))

(sxml-match-let* ([(a ,i ,j) '(a 1 2)])
  (+ i j))

These forms generalize the let and let* forms of Scheme to allow an XML pattern in the binding position, rather than a simple variable.

For example, the expression below:

(sxml-match-let (([a ,i ,j] '(a 1 2)))
  (+ i j))

binds the variables i and j to 1 and 2 in the XML value given.

7.17 The Scheme shell (scsh)
An incomplete port of the Scheme shell (scsh) was once available for Guile as a separate package. However this code has bitrotten somewhat. The pieces are available in Guile's legacy CVS repository, which may be browsed at http://cvs.savannah.gnu.org/viewvc/guile/guile-scsh/?root=guile.

For information about scsh see http://www.scsh.net/.

This bitrotting is a bit of a shame, as there is a good deal of well-written Scheme code in scsh. Adopting this code and porting it to current Guile should be an educational experience, in addition to providing something of value to Guile folks.

7.18 Curried Definitions
The macros in this section are provided by

(use-modules (ice-9 curried-definitions))
Prior to Guile 2.0, Guile provided a type of definition known colloquially as a “curried definition”. The idea is to extend the syntax of \texttt{define} so that you can conveniently define procedures that return procedures, up to any desired depth.

For example,

\begin{verbatim}
(define ((foo x) y)
  (list x y))
\end{verbatim}

is a convenience form of

\begin{verbatim}
(define foo
  (lambda (x)
    (lambda (y)
      (list x y))))
\end{verbatim}

\texttt{define} is similar but the formal parameter lists take additional options as described in Section 6.9.4.1 [lambda* and define*], page 252. For example,

\begin{verbatim}
(define* ((foo #:keys (bar 'baz) (quux 'zot)) frotz #:rest rest)
  (list bar quux frotz rest))
\end{verbatim}

\begin{verbatim}
((foo #:quux 'foo) 1 2 3 4 5)
⇒ (baz foo 1 (2 3 4 5))
\end{verbatim}

\texttt{define-public} is similar to \texttt{define} but it also adds \texttt{name} to the list of exported bindings of the current module.

\section*{7.19 Statprof}

Statprof is a statistical profiler for Guile.

A simple use of statprof would look like this:

\begin{verbatim}
(use-modules (statprof))
(statprof (lambda ()
  (map 1+ (iota 100000))
  #f))
\end{verbatim}

This would run the thunk with statistical profiling, finally displaying a flat table of statistics which could look something like this:

\begin{verbatim}
% cumulative self
time  seconds  seconds procedure
57.14 39769.73 0.07 ice-9/boot-9.scm:249:5:map1
28.57 0.04 0.04 ice-9/boot-9.scm:1165:0:iota
14.29 0.02 0.02 1+
\end{verbatim}
Sample count: 7
Total time: 0.123490713 seconds (0.201983993 seconds in GC)

All of the numerical data with the exception of the calls column is statistically approximate. In the following column descriptions, and in all of statprof, “time” refers to execution time (both user and system), not wall clock time.

The % time column indicates the percentage of the run-time time spent inside the procedure itself (not counting children). It is calculated as self seconds, measuring the amount of time spent in the procedure, divided by the total run-time.

cumulative seconds also counts time spent in children of a function. For recursive functions, this can exceed the total time, as in our example above, because each activation on the stack adds to the cumulative time.

Finally, the GC time measures the time spent in the garbage collector. On systems with multiple cores, this time can be larger than the run time, because it counts time spent in all threads, and will run the “marking” phase of GC in parallel. If GC time is a significant fraction of the run time, that means that most time in your program is spent allocating objects and cleaning up after those allocations. To speed up your program, one good place to start would be to look at how to reduce the allocation rate.

Statprof’s main mode of operation is as a statistical profiler. However statprof can also run in a “precise” mode as well. Pass the #:count-calls? #t keyword argument to statprof to record all calls:

```
(use-modules (statprof))
(statprof (lambda ()
  (map 1+ (iota 1000000))
  #f)
 #:count-calls? #t)
```

The result has an additional calls column:

```
%    cumulative     self
  time  seconds  seconds  calls  procedure
 82.26   0.73     0.73    1000000  1+
 11.29 420925.80  0.10 10000001 ice-9/boot-9.scm:249:5:map1
  4.84   0.06     0.04     1     ice-9/boot-9.scm:1165:0:iota
[...]
```

Sample count: 62
Total time: 0.893098065 seconds (1.222796536 seconds in GC)

As you can see, the profile is perturbed: 1+ ends up on top, whereas it was not marked as hot in the earlier profile. This is because the overhead of call-counting unfairly penalizes calls. Still, this precise mode can be useful at times to do algorithmic optimizations based on the precise call counts.

**Implementation notes**

The profiler works by setting the unix profiling signal ITIMER_PROF to go off after the interval you define in the call to statprof-reset. When the signal fires, a sampling routine runs
which crawls up the stack, recording all instruction pointers into a buffer. After the sample is complete, the profiler resets profiling timer to fire again after the appropriate interval.

Later, when profiling stops, that log buffer is analyzed to produce the “self seconds” and “cumulative seconds” statistics. A procedure at the top of the stack counts toward “self” samples, and everything on the stack counts towards “cumulative” samples.

While the profiler is running it measures how much CPU time (system and user – which is also what ITIMER_PROF tracks) has elapsed while code has been executing within the profiler. Only run time counts towards the profile, not wall-clock time. For example, sleeping and waiting for input or output do not cause the timer clock to advance.

Usage

**statprof** thunk [#:loop loop=1] [#:hz hz=100] [#:port (current-output-port)] [#:count-calls? count-calls?=#f] [#:display-style display-style='flat']

Profile the execution of thunk, and return its return values.

The stack will be sampled hz times per second, and the thunk itself will be called loop times.

If count-calls? is true, all procedure calls will be recorded. This operation is somewhat expensive.

After the thunk has been profiled, print out a profile to port. If display-style is flat, the results will be printed as a flat profile. Otherwise if display-style is tree, print the results as a tree profile.

Note that statprof requires a working profiling timer. Some platforms do not support profiling timers. (provided? 'ITIMER_PROF) can be used to check for support of profiling timers.

Profiling can also be enabled and disabled manually.

**statprof-active?**

Returns #t if statprof-start has been called more times than statprof-stop, #f otherwise.

**statprof-start**

Start or stop the profiler.

**statprof-stop**

Start or stop the profiler.

**statprof-reset sample-seconds sample-microseconds count-calls?**

Reset the profiling sample interval to sample-seconds and sample-microseconds. If count-calls? is true, arrange to instrument procedure calls as well as collecting statistical profiling data.

If you use the manual statprof-start/statprof-stop interface, an implicit statprof state will persist starting from the last call to statprof-reset, or the first call to statprof-start. There are a number of accessors to fetch statistics from this implicit state.
statprof-accumulated-time [Scheme Procedure]
Returns the time accumulated during the last statprof run.

statprof-sample-count [Scheme Procedure]
Returns the number of samples taken during the last statprof run.

statprof-fold-call-data proc init [Scheme Procedure]
Fold proc over the call-data accumulated by statprof. This procedure cannot be called
while statprof is active.
proc will be called with arguments, call-data and prior-result.

statprof-proc-call-data proc [Scheme Procedure]
Returns the call-data associated with proc, or #f if none is available.

statprof-call-data-name cd [Scheme Procedure]
statprof-call-data-calls cd [Scheme Procedure]
statprof-call-data-cum-samples cd [Scheme Procedure]
statprof-call-data-self-samples cd [Scheme Procedure]
Accessors for the fields in a statprof call-data object.

statprof-call-data->stats call-data [Scheme Procedure]
Returns an object of type statprof-stats.

statprof-stats-proc-name stats [Scheme Procedure]
statprof-stats-%-time-in-proc stats [Scheme Procedure]
statprof-stats-cum-secs-in-proc stats [Scheme Procedure]
statprof-stats-self-secs-in-proc stats [Scheme Procedure]
statprof-stats-calls stats [Scheme Procedure]
statprof-stats-self-secs-per-call stats [Scheme Procedure]
statprof-stats-cum-secs-per-call stats [Scheme Procedure]
Accessors for the fields in a statprof-stats object.

statprof-display [port=(current-output-port)] [#:style
style=flat]
Displays a summary of the statistics collected. Possible values for style include:
flat Display a traditional gprof-style flat profile.
anomalies Find statistical anomalies in the data.
tree Display a tree profile.

statprof-fetch-stacks [Scheme Procedure]
Returns a list of stacks, as they were captured since the last call to statprof-reset.

statprof-fetch-call-tree [#:precise precise?=#:f]
Return a call tree for the previous statprof run.

The return value is a list of nodes. A node is a list of the form:
@code
node ::= (@var{proc} @var{count} . @var{nodes})
@end code

The @var{proc} is a printable representation of a procedure, as a string. If @var{precise?} is false, which is the default, then a node corresponds to a procedure invocation. If it is true, then a node corresponds to a return point in a procedure. Passing @code{#:precise? #t} allows a user to distinguish different source lines in a procedure, but usually it is too much detail, so it is off by default.

gcprof thunk [#:loop]  [Scheme Procedure]
Like the statprof procedure, but instead of profiling CPU time, we profile garbage collection.
The stack will be sampled soon after every garbage collection during the evaluation of thunk, yielding an approximate idea of what is causing allocation in your program.
Since GC does not occur very frequently, you may need to use the loop parameter, to cause thunk to be called loop times.

7.20 SXML
SXML is a native representation of XML in terms of standard Scheme data types: lists, symbols, and strings. For example, the simple XML fragment:

<parrot type="African Grey"><name>Alfie</name></parrot>

may be represented with the following SXML:

(parrot (@ (type "African Grey")) (name "Alfie"))

SXML is very general, and is capable of representing all of XML. Formally, this means that SXML is a conforming implementation of the http://www.w3.org/TR/xml-infoset/ (XML Information Set) standard.

Guile includes several facilities for working with XML and SXML: parsers, serializers, and transformers.

7.20.1 SXDomain Overview
(This section needs to be written; volunteers welcome.)

7.20.2 Reading and Writing XML
The (sxml simple) module presents a basic interface for parsing XML from a port into the Scheme SXML format, and for serializing it back to text.

(use-modules (sxml simple))

xml->sxml [string-or-port] [#:namespaces='()]  [Scheme Procedure]
[#:declare-namespaces?=#:t] [#:trim-whitespace=?#f] [#:entities='()]
[#:default-entity-handler=#f] [#:doctype-handler=#f]
Use SSAX to parse an XML document into SXML. Takes one optional argument, string-or-port, which defaults to the current input port. Returns the resulting SXML document. If string-or-port is a port, it will be left pointing at the next available character in the port.
As is normal in SXML, XML elements parse as tagged lists. Attributes, if any, are placed after the tag, within an @ element. The root of the resulting XML will be contained in a special tag, *TOP*. This tag will contain the root element of the XML, but also any prior processing instructions.

```
(xml->sxml "<foo/>")
⇒ (*TOP* (foo))
```

```
(xml->sxml "<foo>text</foo>")
⇒ (*TOP* (foo "text"))
```

```
(xml->sxml "<foo kind="bar">text</foo>")
⇒ (*TOP* (foo (@ (kind "bar")) "text"))
```

```
(xml->sxml "<?xml version="1.0"?><foo/>")
⇒ (*TOP* (*PI* xml "version="1.0"") (foo))
```

All namespaces in the XML document must be declared, via xmlns attributes. SXML elements built from non-default namespaces will have their tags prefixed with their URI. Users can specify custom prefixes for certain namespaces with the #:namespaces keyword argument to xml->sxml.

```
(xml->sxml "<foo xmlns="http://example.org/ns1">text</foo>"
#:namespaces '((ns1 . "http://example.org/ns1")))
⇒ (*TOP* (ns1:foo "text"))
```

```
(xml->sxml "<foo xmlns:bar="http://example.org/ns2"><bar:baz/></foo>
#:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
```

By default, namespaces passed to xml->sxml are treated as if they were declared on the root element. Passing a false #:declare-namespaces? argument will disable this behavior, requiring in-document declarations of namespaces before use.

```
(xml->sxml "<foo><ns2:baz/></foo>
#:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
```

```
(error: undeclared namespace: ‘bar’)
```

By default, all whitespace in XML is significant. Passing the #:trim-whitespace? keyword argument to xml->sxml will trim whitespace in front, behind and between elements, treating it as “unsignificant”. Whitespace in text fragments is left alone.

```
(xml->sxml "<foo><ns2:baz/></foo>
#:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
```

```
(error: undeclared namespace: ‘bar’)
```

```
(xml->sxml "<foo><bar> Alfie the parrot! </bar><foo/>")
⇒ (*TOP* (foo (bar " Alfie the parrot! ") ("\n")))
```

```
(xml->sxml "<foo><bar> Alfie the parrot! </bar><foo>
#:trim-whitespace? #t)
⇒ (*TOP* (foo (bar " Alfie the parrot! ")))
```

Parsed entities may be declared with the #:entities keyword argument, or handled with the #:default-entity-handler. By default, only the standard &lt;, &gt;, &amp;, &apos;
and "quote"; entities are defined, as well as the \&\#N; and \&\#xN; (decimal and hexadecimal) numeric character entities.

(xml->sxml "<foo>&amp;</foo>")
⇒ (*TOP* (foo "&"))

(xml->sxml "<foo>&nbsp;</foo>")
⇒ error: undefined entity: nbsp

(xml->sxml "<foo>&#xA0;</foo>")
⇒ (*TOP* (foo "\xa0"))

(xml->sxml "<foo>&nbsp; &foo;</foo>
 #:entities '((nbsp . \"\xa0\")))
⇒ (*TOP* (foo "\xa0"))

(xml->sxml "<foo>&nbsp; &foo;</foo>
 #:default-entity-handler
 (lambda (port name)
   (case name
     ((nbsp) "\xa0")
     (else
      (format (current-warning-port)
        "\""a:`a:`a: undefined entity: `a\n" (or (port-filename port) "<unknown file>")
        (port-line port) (port-column port)
        name)
      (symbol->string name))))))
⇒ <unknown file>:0:17: undefined entity: foo
⇒ (*TOP* (foo "\xa0 foo"))

By default, xml->sxml skips over the <!DOCTYPE> declaration, if any. This behavior can be overridden with the #:doctype-handler argument, which should be a procedure of three arguments: the docname (a symbol), systemid (a string), and the internal doctype subset (as a string or #f if not present).

The handler should return keyword arguments as multiple values, as if it were calling its continuation with keyword arguments. The continuation accepts the #:entities and #:namespaces keyword arguments, in the same format that xml->sxml itself takes. These entities and namespaces will be prepended to those given to the xml->sxml invocation.

(define (handle-foo docname systemid internal-subset)
  (case docname
    ((foo)
     (values #:entities '(((greets . "<i>Hello, world!</i>")))))
    (else
     (values))))

(xml->sxml "<!DOCTYPE foo><p>&greets;</p>
 #:doctype-handler handle-foo)
⇒ (*TOP* (p (i "Hello, world!")))

If the document has no doctype declaration, the doctype-handler is invoked with #f for the three arguments.
In the future, the continuation may accept other keyword arguments, for example to validate the parsed SXML against the doctype.

sxml->xml tree [port]  
(Scheme Procedure)
Serialize the SXML tree tree as XML. The output will be written to the current output port, unless the optional argument port is present.

sxml->string sxml  
(Scheme Procedure)
Detag an sxml tree sxml into a string. Does not perform any formatting.

7.20.3 SSAX: A Functional XML Parsing Toolkit
Guile’s XML parser is based on Oleg Kiselyov’s powerful XML parsing toolkit, SSAX.

7.20.3.1 History
Back in the 1990s, when the world was young again and XML was the solution to all of its problems, there were basically two kinds of XML parsers out there: DOM parsers and SAX parsers.

A DOM parser reads through an entire XML document, building up a tree of “DOM objects” representing the document structure. They are very easy to use, but sometimes you don’t actually want all of the information in a document; building an object tree is not necessary if all you want to do is to count word frequencies in a document, for example.

SAX parsers were created to give the programmer more control on the parsing process. A programmer gives the SAX parser a number of “callbacks”: functions that will be called on various features of the XML stream as they are encountered. SAX parsers are more efficient, but much harder to user, as users typically have to manually maintain a stack of open elements.

Kiselyov realized that the SAX programming model could be made much simpler if the callbacks were formulated not as a linear fold across the features of the XML stream, but as a tree fold over the structure implicit in the XML. In this way, the user has a very convenient, functional-style interface that can still generate optimal parsers.

The xml->sxml interface from the (sxml simple) module is a DOM-style parser built using SSAX, though it returns SXML instead of DOM objects.

7.20.3.2 Implementation
(sxml ssax) is a package of low-to-high level lexing and parsing procedures that can be combined to yield a SAX, a DOM, a validating parser, or a parser intended for a particular document type. The procedures in the package can be used separately to tokenize or parse various pieces of XML documents. The package supports XML Namespaces, internal and external parsed entities, user-controlled handling of whitespace, and validation. This module therefore is intended to be a framework, a set of “Lego blocks” you can use to build a parser following any discipline and performing validation to any degree. As an example of the parser construction, the source file includes a semi-validating SXML parser.

SSAX has a “sequential” feel of SAX yet a “functional style” of DOM. Like a SAX parser, the framework scans the document only once and permits incremental processing. An application that handles document elements in order can run as efficiently as possible.
Unlike a SAX parser, the framework does not require an application register stateful call-
backs and surrender control to the parser. Rather, it is the application that can drive the
framework – calling its functions to get the current lexical or syntax element. These func-
tions do not maintain or mutate any state save the input port. Therefore, the framework
permits parsing of XML in a pure functional style, with the input port being a monad (or
a linear, read-once parameter).

Besides the port, there is another monad – seed. Most of the middle- and high-level
parsers are single-threaded through the seed. The functions of this framework do not
process or affect the seed in any way: they simply pass it around as an instance of an
opaque datatype. User functions, on the other hand, can use the seed to maintain user’s
state, to accumulate parsing results, etc. A user can freely mix their own functions with
those of the framework. On the other hand, the user may wish to instantiate a high-level
parser: SSAX:make-elem-parser or SSAX:make-parser. In the latter case, the user must
provide functions of specific signatures, which are called at predictable moments during
the parsing: to handle character data, element data, or processing instructions (PI). The
functions are always given the seed, among other parameters, and must return the new
seed.

From a functional point of view, XML parsing is a combined pre-post-order traversal of
a “tree” that is the XML document itself. This down-and-up traversal tells the user about
an element when its start tag is encountered. The user is notified about the element once
more, after all element’s children have been handled. The process of XML parsing therefore
is a fold over the raw XML document. Unlike a fold over trees defined in [1], the parser is
necessarily single-threaded – obviously as elements in a text XML document are laid down
sequentially. The parser therefore is a tree fold that has been transformed to accept an
accumulating parameter [1,2].

Formally, the denotational semantics of the parser can be expressed as

\[
\text{parser} :: (\text{Start-tag} \rightarrow \text{Seed} \rightarrow \text{Seed}) \rightarrow \\
(\text{Start-tag} \rightarrow \text{Seed} \rightarrow \text{Seed} \rightarrow \text{Seed}) \rightarrow \\
(\text{Char-Data} \rightarrow \text{Seed} \rightarrow \text{Seed}) \rightarrow \\
\text{XML-text-fragment} \rightarrow \text{Seed} \rightarrow \text{Seed}
\]

\[
\text{parser fdown fup fchar "<elem attrs> content </elem>" seed} \\
= \text{fup "<elem attrs>" seed} \\
\text{(parser fdown fup fchar "content" (fdown "<elem attrs>" seed))}
\]

\[
\text{parser fdown fup fchar "char-data content" seed} \\
= \text{parser fdown fup fchar "content" (fchar "char-data" seed)}
\]

\[
\text{parser fdown fup fchar "elem-content content" seed} \\
= \text{parser fdown fup fchar "content" (fchar "elem-content" seed)}
\]

Compare the last two equations with the left fold

\[
\text{fold-left kons elem:list seed} = \text{fold-left kons list (kons elem seed)}
\]

The real parser created by SSAX:make-parser is slightly more complicated, to account
for processing instructions, entity references, namespaces, processing of document type
declaration, etc.

The XML standard document referred to in this module is http://www.w3.org/TR/
1998/REC-xml-19980210.html
The present file also defines a procedure that parses the text of an XML document or of a separate element into SXML, an S-expression-based model of an XML Information Set. SXML is also an Abstract Syntax Tree of an XML document. SXML is similar but not identical to DOM; SXML is particularly suitable for Scheme-based XML/HTML authoring, SXPath queries, and tree transformations. See SXML.html for more details. SXML is a term implementation of evaluation of the XML document [3]. The other implementation is context-passing.

The present frameworks fully supports the XML Namespaces Recommendation: http://www.w3.org/TR/REC-xml-names/.

Other links:

7.20.3.3 Usage

```scheme
(current-ssax-error-port) [Scheme Procedure]
(with-ssax-error-to-port port thunk) [Scheme Procedure]
(xml-token? _ ) [Scheme Procedure]
   -- Scheme Procedure: pair? x
   Return '#t' if X is a pair; otherwise return '#f'.

(xml-token-kind token) [Scheme Syntax]
(xml-token-head token) [Scheme Syntax]
(make-empty-attlist) [Scheme Procedure]
(attlist-add attlist name-value) [Scheme Procedure]
(attlist-null? x) [Scheme Procedure]
   Return #t if x is the empty list, else #f.

(attlist-remove-top attlist) [Scheme Procedure]
(attlist->alist attlist) [Scheme Procedure]
(attlist-fold kons knil lis1) [Scheme Procedure]
(define-parsed-entity! entity str) [Scheme Procedure]
   Define a new parsed entity. entity should be a symbol.
   Instances of &entity; in XML text will be replaced with the string str, which will then be parsed.
```
reset-parsed-entity-definitions!
   Restore the set of parsed entity definitions to its initial state.

ssax:uri-string->symbol uri-str

ssax:skip-internal-dtd port

ssax:read-pi-body-as-string port

ssax:reverse-collect-str-drop-ws fragments

ssax:read-markup-token port

ssax:read-cdata-body port str-handler seed

ssax:read-char-ref port

ssax:read-attributes port entities

ssax:complete-start-tag tag-head port elems entities namespaces

ssax:read-external-id port

ssax:read-char-data port expect-eof? str-handler seed

ssax:xml->sxml port namespace-prefix-assig

ssax:make-parser . kw-val-pairs

ssax:make-pi-parser orig-handlers

ssax:make-elem-parser my-new-level-seed my-finish-element
   my-char-data-handler my-pi-handlers

7.20.4 Transforming SXML

7.20.4.1 Overview

SXML expression tree transformers

Pre-Post-order traversal of a tree and creation of a new tree

\[
\text{pre-post-order}:: \langle\text{tree}\rangle \times \langle\text{bindings}\rangle \rightarrow \langle\text{new-tree}\rangle
\]

where

\[
\langle\text{bindings}\rangle ::= \langle\text{binding}\rangle \ldots
\]

\[
\langle\text{binding}\rangle ::= \langle\text{trigger-symbol}\rangle \ast \text{preorder}\ast \ . \langle\text{handler}\rangle\rangle \mid
\langle\text{trigger-symbol}\rangle \ast \text{macro}\ast \ . \langle\text{handler}\rangle\rangle \mid
\langle\text{trigger-symbol}\rangle \ast \text{new-bindings}\ . \langle\text{handler}\rangle\rangle \mid
\langle\text{trigger-symbol}\rangle \ast \langle\text{handler}\rangle\rangle
\]

The pre-post-order function visits the nodes and nodelists pre-post-order (depth-first). For each \text{Node} of the form \langle\text{name} \langle\text{Node}\rangle \ldots\rangle, it looks up an association with the given \text{name} among its \langle\text{bindings}\rangle. If failed, \text{pre-post-order} tries to locate a \ast\text{default}\ast binding. It’s an error if the latter attempt fails as well. Having found a binding, the \text{pre-post-order} function first checks to see if the binding is of the form

\[
\langle\text{trigger-symbol}\rangle \ast \text{preorder}\ast \ . \langle\text{handler}\rangle\rangle
\]
If it is, the handler is 'applied' to the current node. Otherwise, the pre-post-order function first calls itself recursively for each child of the current node, with `<new-bindings>` prepended to the `<bindings>` in effect. The result of these calls is passed to the `<handler>` (along with the head of the current `<Node>`). To be more precise, the handler is _applied_ to the head of the current node and its processed children. The result of the handler, which should also be a `<tree>`, replaces the current `<Node>`. If the current `<Node>` is a text string or other atom, a special binding with a symbol *text* is looked up.

A binding can also be of a form

```scheme
(<trigger-symbol> *macro* . <handler>)
```

This is equivalent to *preorder* described above. However, the result is re-processed again, with the current stylesheet.

### 7.20.4.2 Usage

- **SRV:send-reply** . `fragments`  
  Output the `fragments` to the current output port.
  
  The fragments are a list of strings, characters, numbers, thunks, #f, #t – and other fragments. The function traverses the tree depth-first, writes out strings and characters, executes thunks, and ignores #f and '(). The function returns #t if anything was written at all; otherwise the result is #f if #t occurs among the fragments, it is not written out but causes the result of SRV:send-reply to be #t.

- **foldts fdown fup flhere seed tree**  
  The standard multithreaded tree fold.
  
  `fup` is of type [a] -> a. `flhere` is of type object -> a.

- **foldts fdown fup flhere seed tree**  
  The single-threaded tree fold originally defined in SSAX. See Section 7.20.3 [SSAX], page 736, for more information.

- **foldts* fdown fup flhere seed tree**  
  A variant of `foldts` that allows pre-order tree rewrites. Originally defined in Andy Wingo’s 2007 paper, *Applications of fold to XML transformation*. 

### 7.20.5 SXML Tree Fold

#### 7.20.5.1 Overview

`(sxml fold)` defines a number of variants of the `fold` algorithm for use in transforming SXML trees. Additionally it defines the layout operator, fold-layout, which might be described as a context-passing variant of SSAX’s pre-post-order.

#### 7.20.5.2 Usage

- **foldt fup flhere tree**  
  The standard multithreaded tree fold.
  
  `fup` is of type [a] -> a. `flhere` is of type object -> a.
fold-values proc list . seeds

A variant of fold that allows multi-valued seeds. Note that the order of the arguments differs from that of fold. See Section 7.5.3.5 [SRFI-1 Fold and Map], page 582.

foldts*-values fdown fup fhere tree . seeds

A variant of foldts* that allows multi-valued seeds. Originally defined in Andy Wingo’s 2007 paper, Applications of fold to XML transformation.

fold-layout tree bindings params layout stylesheet

A traversal combinator in the spirit of pre-post-order. See Section 7.20.4 [Transforming SXML], page 739.

fold-layout was originally presented in Andy Wingo’s 2007 paper, Applications of fold to XML transformation.

bindings := (<binding>...)

binding := (<tag> <handler-pair>...)
  | (*default* . <post-handler>)
  | (*text* . <text-handler>)

tag := <symbol>

handler-pair := (pre-layout . <pre-layout-handler>)
  | (post . <post-handler>)
  | (bindings . <bindings>)
  | (pre . <pre-handler>)
  | (macro . <macro-handler>)

pre-layout-handler

A function of three arguments:

kids the kids of the current node, before traversal

params the params of the current node

layout the layout coming into this node

pre-layout-handler is expected to use this information to return a layout to pass to the kids. The default implementation returns the layout given in the arguments.

post-handler

A function of five arguments:

tag the current tag being processed

params the params of the current node

layout the layout coming into the current node, before any kids were processed

klayout the layout after processing all of the children

kids the already-processed child nodes

post-handler should return two values, the layout to pass to the next node and the final tree.
text-handler

text-handler is a function of three arguments:

- `text`: the string
- `params`: the current params
- `layout`: the current layout

text-handler should return two values, the layout to pass to the next node and the value to which the string should transform.

### 7.20.6 SXPath

#### 7.20.6.1 Overview

**SXPath: SXML Query Language**

SXPath is a query language for SXML, an instance of XML Information set (Infoset) in the form of s-expressions. See (sxml ssax) for the definition of SXML and more details. SXPath is also a translation into Scheme of an XML Path Language, XPath (http://www.w3.org/TR/xpath). XPath and SXPath describe means of selecting a set of Infoset’s items or their properties.

To facilitate queries, XPath maps the XML Infoset into an explicit tree, and introduces important notions of a location path and a current, context node. A location path denotes a selection of a set of nodes relative to a context node. Any XPath tree has a distinguished, root node – which serves as the context node for absolute location paths. Location path is recursively defined as a location step joined with a location path. A location step is a simple query of the database relative to a context node. A step may include expressions that further filter the selected set. Each node in the resulting set is used as a context node for the adjoining location path. The result of the step is a union of the sets returned by the latter location paths.

The SXML representation of the XML Infoset (see SSAX.scn) is rather suitable for querying as it is. Bowing to the XPath specification, we will refer to SXML information items as ‘Nodes’:

```scheme
<Node> ::= <Element> | <attributes-coll> | <attrib>
          | "text string" | <PI>
```

This production can also be described as

```scheme
<Node> ::= (name . <Nodeset>) | "text string"
```

An (ordered) set of nodes is just a list of the constituent nodes:

```scheme
<Nodeset> ::= (<Node> ...)
```

Nodesets, and Nodes other than text strings are both lists. A <Nodeset> however is either an empty list, or a list whose head is not a symbol. A symbol at the head of a node is either an XML name (in which case it’s a tag of an XML element), or an administrative name such as '@'. This uniform list representation makes processing rather simple and elegant, while avoiding confusion. The multi-branch tree structure formed by the mutually-recursive datatypes <Node> and <Nodeset> lends itself well to processing by functional languages.
A location path is in fact a composite query over an XPath tree or its branch. A single step is a combination of a projection, selection or a transitive closure. Multiple steps are combined via join and union operations. This insight allows us to elegantly implement XPath as a sequence of projection and filtering primitives – converters – joined by combinators. Each converter takes a node and returns a nodeset which is the result of the corresponding query relative to that node. A converter can also be called on a set of nodes. In that case it returns a union of the corresponding queries over each node in the set. The union is easily implemented as a list append operation as all nodes in an SXML tree are considered distinct, by XPath conventions. We also preserve the order of the members in the union. Query combinators are high-order functions: they take converter(s) (which is a Node|Nodeset -> Nodeset function) and compose or otherwise combine them. We will be concerned with only relative location paths [XPath]: an absolute location path is a relative path applied to the root node.

Similarly to XPath, SXPath defines full and abbreviated notations for location paths. In both cases, the abbreviated notation can be mechanically expanded into the full form by simple rewriting rules. In the case of SXPath the corresponding rules are given in the documentation of the sxpath procedure. See [XPath procedure documentation], page 746.

The regression test suite at the end of the file SXPATH-old.scm shows a representative sample of SXPaths in both notations, juxtaposed with the corresponding XPath expressions. Most of the samples are borrowed literally from the XPath specification.

Much of the following material is taken from the SXPath sources by Oleg Kiselyov et al.

7.20.6.2 Basic Converters and Applicators

A converter is a function mapping a nodeset (or a single node) to another nodeset. Its type can be represented like this:

\[
\text{type Converter} = \text{Node|Nodeset} \rightarrow \text{Nodeset}
\]

A converter can also play the 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. Likewise, an empty nodeset is equivalent to \#f in denoting failure.

\[
\text{nodeset? } x
\]

Return \#t if \(x\) is a nodeset.

\[
\text{node-typeof? } \text{crit}
\]

This function implements a 'Node test' as defined in Sec. 2.3 of the XPath document. A node test is one of the components of a location step. It is also a converter-predicate in SXPath.

The function node-typeof? takes a type criterion and returns a function, which, when applied to a node, will tell if the node satisfies the test.

The criterion \text{crit} is a symbol, one of the following:

- \text{id} tests if the node has the right name (id)
- \text{@} tests if the node is an \text{<attributes-coll>}
- \text{*} tests if the node is an \text{<Element>}
- \text{*text*} tests if the node is a text node
*PI* tests if the node is a PI (processing instruction) node

*any* #t for any type of node

node-eq? other

A curried equivalence converter predicate that takes a node other and returns a function that takes another node. The two nodes are compared using eq?.

node-equal? other

A curried equivalence converter predicate that takes a node other and returns a function that takes another node. The two nodes are compared using equal?.

node-pos n

Select the n’th element of a nodeset and return as a singular nodeset. If the n’th element does not exist, return an empty nodeset. If n is a negative number the node is picked from the tail of the list.

((node-pos 1) nodeset) ; return the the head of the nodeset (if exists)
((node-pos 2) nodeset) ; return the node after that (if exists)
((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.

filter pred?

A filter applicator, which introduces a filtering context. The argument converter pred? is considered a predicate, with either #f or nil meaning failure.

take-until pred?

take-until:: Converter -> Converter, or
  take-until:: Pred -> Node|Nodeset -> Nodeset

Given a converter-predicate pred? 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 until 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 up to (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?

take-after:: Converter -> Converter, or
  take-after:: Pred -> Node|Nodeset -> Nodeset

Given a converter-predicate pred? 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  
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  
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  
node-trace:: String -> Converter
(node-trace title) is an identity converter. In addition it prints out the node or nodeset it is applied to, prefixed with the title. This converter is very useful for debugging.

7.20.6.3 Converter Combinators

Combinators are 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?
select-kids takes a converter (or a predicate) as an argument and returns another converter. The resulting converter applied to a nodeset returns an ordered subset of its children that satisfy the predicate test-pred?.

node-self pred?
Similar to select-kids except that the predicate pred? is applied 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 predicate.

node-join . selectors  
node-join:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-join:: [Converter] -> Converter
Join the sequence of location steps or paths as described above.

node-reduce . converters  
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**  

```scheme
(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?**  

```scheme
(node-closure:: Converter -> Converter)
```

Select all descendants of a node that satisfy a converter-predicate test-pred?. 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

```scheme
(define (node-closure f)
  (node-or
    (select-kids f)
    (node-reduce (select-kids (node-typeof? '*)) (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 (node-typeof? '*)) will return an empty nodeset. At this point further iterations will no longer affect the result and can be stopped.

**node-parent rootnode**  

```scheme
(node-parent:: RootNode -> Converter)
```

(node-parent rootnode) yields a converter that 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.

Given the notation of Philip Wadler’s paper on semantics of XSLT,

```
parent(x) = { y | y=subnode*(root), x=subnode(y) }
```

Therefore, node-parent is not the fundamental converter: it can be expressed through the existing ones. Yet node-parent is a rather convenient converter. It corresponds to a parent:: axis of SXPath. Note that the parent:: axis can be used with an attribute node as well.

**xpath path**  

```scheme
(xpath:: AbbrPath -> Converter, or
  xpath:: AbbrPath -> Node|Nodeset -> Nodeset)
```

path is a list. It is translated to the full SXPath according to the following rewriting rules:

```scheme
(sxpath '())
⇒ (node-join)
(xpath '(path-component ...))
⇒ (node-join (xpath1 path-component) (xpath '(...)))

(xpath1 '//)
⇒ (node-or
   (node-self (node-typeof? '*any*))
   (node-closure (node-typeof? '*any*)))

(xpath1 '(equal? x))
⇒ (select-kids (node-equal? x))

(xpath1 '(eq? x))
⇒ (select-kids (node-eq? x))

(xpath1 ?symbol)
⇒ (select-kids (node-typeof? ?symbol)

(xpath1 procedure)
⇒ procedure

(xpath1 '(?symbol ...))
⇒ (xpath1 '((?symbol) ...))

(xpath1 '(path reducer ...))
⇒ (node-reduce (xpath path) (xpathr reducer) ...)

(xpathr number)
⇒ (node-pos number)

(xpathr path-filter)
⇒ (filter (xpath path-filter))

7.20.7 (sxml ssax input-parse)

7.20.7.1 Overview
A simple lexer.

The procedures in this module surprisingly often suffice to parse an input stream. They either skip, or build and return tokens, according to inclusion or delimiting semantics. The list of characters to expect, include, or to break at may vary from one invocation of a function to another. This allows the functions to easily parse even context-sensitive languages.

EOF is generally frowned on, and thrown up upon if encountered. Exceptions are mentioned specifically. The list of expected characters (characters to skip until, or break characters) may include an EOF "character", which is to be coded as the symbol, *eof*. 

The input stream to parse is specified as a port, which is usually the last (and optional) argument. It defaults to the current input port if omitted.
If the parser encounters an error, it will throw an exception to the key `parser-error`. The arguments will be of the form `(port message specialising-msg*)`.

The first argument is a port, which typically points to the offending character or its neighborhood. You can then use `port-column` and `port-line` to query the current position. `message` is the description of the error. Other arguments supply more details about the problem.

### 7.20.7.2 Usage

- `peek-next-char [port]` [Scheme Procedure]
- `assert-curr-char expected-chars comment [port]` [Scheme Procedure]
- `skip-until arg [port]` [Scheme Procedure]
- `skip-while skip-chars [port]` [Scheme Procedure]
- `next-token prefix-skipped-chars break-chars [comment] [port]` [Scheme Procedure]
- `next-token-of incl-list/pred [port]` [Scheme Procedure]
- `read-text-line [port]` [Scheme Procedure]
- `read-string n [port]` [Scheme Procedure]
- `find-string-from-port? str in <input-port>, optionally within the first max-no-char characters.` [Scheme Procedure]

### 7.20.8 (sxml apply-templates)

#### 7.20.8.1 Overview

Pre-order traversal of a tree and creation of a new tree:

```
apply-templates:: tree x <templates> -> <new-tree>
```

where

```
<templates> ::= (<template> ...)
<template> ::= (<node-test> <node-test> ... <node-test> . <handler>)
<node-test> ::= an argument to node-typeof? above
<handler> ::= <tree> -> <new-tree>
```

This procedure does a normal, pre-order traversal of an SXML tree. It walks the tree, checking at each node against the list of matching templates.

If the match is found (which must be unique, i.e., unambiguous), the corresponding handler is invoked and given the current node as an argument. The result from the handler, which must be a `<tree>`, takes place of the current node in the resulting tree. The name of the function is not accidental: it resembles rather closely an `apply-templates` function of XSLT.

#### 7.20.8.2 Usage

```
apply-templates tree templates` [Scheme Procedure]
7.21 Texinfo Processing

7.21.1 (texinfo)

7.21.1.1 Overview

Texinfo processing in scheme

This module parses texinfo into SXML. TeX will always be the processor of choice for print output, of course. However, although makeinfo works well for info, its output in other formats is not very customizable, and the program is not extensible as a whole. This module aims to provide an extensible framework for texinfo processing that integrates texinfo into the constellation of SXML processing tools.

Notes on the SXML vocabulary

Consider the following texinfo fragment:

@deffn Primitive set-car! pair value
   This function...
@end deffn

Logically, the category (Primitive), name (set-car!), and arguments (pair value) are “attributes” of the deffn, with the description as the content. However, texinfo allows for @-commands within the arguments to an environment, like @deffn, which means that texinfo “attributes” are PCDATA. XML attributes, on the other hand, are CDATA. For this reason, “attributes” of texinfo @-commands are called “arguments”, and are grouped under the special element, '%'.

Because '%' is not a valid NCName, stexinfo is a superset of SXML. In the interests of interoperability, this module provides a conversion function to replace the '%' with ‘texinfo-arguments’.

7.21.1.2 Usage

call-with-file-and-dir filename proc
   Call the one-argument procedure proc with an input port that reads from filename. During the dynamic extent of proc's execution, the current directory will be (dirname filename). This is useful for parsing documents that can include files by relative path name.

[Function]
texi-command-specs
[Variable]
texi-command-depth command max-depth
   Given the texinfo command command, return its nesting level, or #f if it nests too deep for max-depth.

Examples:

(texi-command-depth 'chapter 4) ⇒ 1
(texi-command-depth 'top 4) ⇒ 0
(texi-command-depth 'subsection 4) ⇒ 3
(texi-command-depth 'appendixsubsec 4) ⇒ 3
(texi-command-depth 'subsection 2) ⇒ #f
texi-fragment->stexi string-or-port  [Function]
Parse the texinfo commands in string-or-port, and return the resultant stexi tree.
The head of the tree will be the special command, *fragment*.

texi->stexi port  [Function]
Read a full texinfo document from port and return the parsed stexi tree. The parsing
will start at the @settitle and end at @bye or EOF.

stexi->sxml tree  [Function]
Transform the stexi tree tree into sxml. This involves replacing the \% element that
keeps the texinfo arguments with an element for each argument.

FIXME: right now it just changes \% to texinfo-arguments – that doesn’t hang with
the idea of making a dtd at some point

7.21.2 (texinfo docbook)

7.21.2.1 Overview
This module exports procedures for transforming a limited subset of the SXML representa-
tion of docbook into stexi. It is not complete by any means. The intention is to gather
a number of routines and stylesheets so that external modules can parse specific subsets of
docbook, for example that set generated by certain tools.

7.21.2.2 Usage

*sdocbook->stexi-rules*  [Variable]
*sdocbook-block-commands*  [Variable]

sdocbook-flatten sdocbook  [Function]
"Flatten" a fragment of sdocbook so that block elements do not nest inside each
other.
Docbook is a nested format, where e.g. a refsect2 normally appears inside a
refsect1. Logical divisions in the document are represented via the tree topology; a
refsect2 element contains all of the elements in its section.
On the contrary, texinfo is a flat format, in which sections are marked off by standalone
section headers like @subsection, and block elements do not nest inside each other.
This function takes a nested sdocbook fragment sdocbook and flattens all of the
sections, such that e.g.

(refsect1 (refsect2 (para "Hello")))

becomes

((refsect1) (refsect2) (para "Hello"))

Oftentimes (always?) sectioning elements have <title> as their first element child;
users interested in processing the refsect* elements into proper sectioning elements
like chapter might be interested in replace-titles and filter-empty-elements.
See [replace-titles], page 751, and [filter-empty-elements], page 751.

Returns a nodeset; that is to say, an untagged list of stexi elements. See Section 7.20.6
[SXPath], page 742, for the definition of a nodeset.
filter-empty-elements sdocbook  
Filters out empty elements in an sdocbook nodeset. Mostly useful after running sdocbook-flatten.

replace-titles sdocbook-fragment  
Iterate over the sdocbook nodeset sdocbook-fragment, transforming contiguous refsect and title elements into the appropriate texinfo sectioning command. Most useful after having run sdocbook-flatten.

For example:

\[(\text{replace-titles } '((\text{refsect1}) (\text{title }"\text{Foo}"\text{)} (\text{para }"\text{Bar.}"\text{))))\]  
\[\Rightarrow '((\text{chapter }"\text{Foo}"\text{)} (\text{para }"\text{Bar."}))\]

7.21.3 (texinfo html)

7.21.3.1 Overview

This module implements transformation from stexi to HTML. Note that the output of stexi->shtml is actually SXML with the HTML vocabulary. This means that the output can be further processed, and that it must eventually be serialized by sxm1->xml. See Section 7.20.2 [Reading and Writing XML], page 733.

References (i.e., the @ref family of commands) are resolved by a ref-resolver. See [texinfo html add-ref-resolver!], page 751.

7.21.3.2 Usage

add-ref-resolver! proc  
Add proc to the head of the list of ref-resolvers. proc will be expected to take the name of a node and the name of a manual and return the URL of the referent, or #f to pass control to the next ref-resolver in the list.

The default ref-resolver will return the concatenation of the manual name, #, and the node name.

stexi->shtml tree  
Transform the stexi tree into shtml, resolving references via ref-resolvers. See the module commentary for more details.

urlify str  

7.21.4 (texinfo indexing)

7.21.4.1 Overview

Given a piece of stexi, return an index of a specified variety.

Note that currently, stexi-extract-index doesn’t differentiate between different kinds of index entries. That’s a bug ;)

7.21.4.2 Usage

stexi-extract-index tree manual-name kind  
Given an stexi tree tree, index all of the entries of type kind. kind can be one of the predefined texinfo indices (concept, variable, function, key, program, type) or
one of the special symbols auto or all. auto will scan the stext for a (printindex) statement, and all will generate an index from all entries, regardless of type.

The returned index is a list of pairs, the car of which is the entry (a string) and the cdr of which is a node name (a string).

7.21.5 (texinfo string-utils)

7.21.5.1 Overview

Module `(texinfo string-utils)` provides various string-related functions useful to Guile’s texinfo support.

7.21.5.2 Usage

**escape-special-chars** str special-chars escape-char

Returns a copy of str with all given special characters preceded by the given escape-char.

special-chars can either be a single character, or a string consisting of all the special characters.

```lisp
;; make a string regexp-safe...
(escape-special-chars "***\(Example String\)***"
 "\[\]/\.*\#

=> "\[\]/\.*\#

;; also can escape a singe char...
(escape-special-chars "richardt@vzavenue.net"
 \#\@\#\@)

=> "richardt@vzavenue.net"
```

**transform-string** str match? replace [start] [end]

Uses match? against each character in str, and performs a replacement on each character for which matches are found.

match? may either be a function, a character, a string, or #t. If match? is a function, then it takes a single character as input, and should return #t for matches. match? is a character, it is compared to each string character using char=? . If match? is a string, then any character in that string will be considered a match. #t will cause every character to be a match.

If replace is a function, it is called with the matched character as an argument, and the returned value is sent to the output string via ‘display’. If replace is anything else, it is sent through the output string via ‘display’.

Note that the replacement for the matched characters does not need to be a single character. That is what differentiates this function from ‘string-map’, and what makes it useful for applications such as converting ‘#\&’ to ‘"&’ in web page text. Some other functions in this module are just wrappers around common uses of ‘transform-string’. Transformations not possible with this function should probably be done with regular expressions.
If \( \text{start} \) and \( \text{end} \) are given, they control which portion of the string undergoes transformation. The entire input string is still output, though. So, if \( \text{start} \) is ‘5’, then the first five characters of \( 	ext{str} \) will still appear in the returned string.

\[
; \text{these two are equivalent...}
\]

\[
(\text{transform-string } \text{str} \ #\text{\space #\-}) ; \text{change all spaces to } \text{-’s}
\]

\[
(\text{transform-string } \text{str} \ (\lambda (\text{c}) (\text{char=? #\space c})) \ #\-)
\]

\[
\text{expand-tabs } \text{str} \ [\text{tab-size}]
\]

Returns a copy of \( \text{str} \) with all tabs expanded to spaces. \( \text{tab-size} \) defaults to 8.

Assuming tab size of 8, this is equivalent to:

\[
(\text{transform-string } \text{str} \ #\text{\tab "
"
})
\]

\[
\text{center-string } \text{str} \ [\text{width}] \ [\text{chr}] \ [\text{rchr}]
\]

Returns a copy of \( \text{str} \) centered in a field of \( \text{width} \) characters. Any needed padding is done by character \( \text{chr} \), which defaults to ‘#\space’. If \( \text{rchr} \) is provided, then the padding to the right will use it instead. See the examples below. Left and \( \text{rchr} \) on the right. The default \( \text{width} \) is 80. The default \( \text{chr} \) and \( \text{rchr} \) is ‘#\space’. The string is never truncated.

\[
(\text{center-string } "\text{Richard Todd}" \ 24)
=> "    Richard Todd    
"
\]

\[
(\text{center-string } "\text{Richard Todd}" \ 24 \ #\=)
=> "====== Richard Todd ======
"
\]

\[
(\text{center-string } "\text{Richard Todd}" \ 24 \ #\< \ #\>)
=> "<<<<<< Richard Todd >>>>>
"
\]

\[
\text{left-justify-string } \text{str} \ [\text{width}] \ [\text{chr}]
\]

\[
\text{left-justify-string } \text{str} \ [\text{width} \ \text{chr}]. \text{Returns a copy of } \text{str} \text{ padded with } \text{chr} \text{ such that it is left justified in a field of } \text{width} \text{ characters. The default } \text{width} \text{ is 80. Unlike ‘string-pad’ from srfi-13, the string is never truncated.}
\]

\[
\text{right-justify-string } \text{str} \ [\text{width}] \ [\text{chr}]
\]

\[
\text{right-justify-string } \text{str} \ [\text{width} \ \text{chr}]. \text{Returns a copy of } \text{str} \text{ padded with } \text{chr} \text{ such that it is right justified in a field of } \text{width} \text{ characters. The default } \text{width} \text{ is 80. The default } \text{chr} \text{ is ‘#\space’. Unlike ‘string-pad’ from srfi-13, the string is never truncated.}
\]

\[
\text{collapse-repeated-chars } \text{str} \ [\text{chr}] \ [\text{num}]
\]

\[
\text{collapse-repeated-chars } \text{str} \ [\text{chr}]. \text{Returns a copy of } \text{str} \text{ with all repeated instances of } \text{chr} \text{ collapsed down to at most } \text{num} \text{ instances. The default value for } \text{chr} \text{ is ‘#\space’, and the default value for } \text{num} \text{ is 1.}
\]

\[
(\text{collapse-repeated-chars } "\text{H e l l o}"
)=> "\text{H e l l o}"
\]

\[
(\text{collapse-repeated-chars } "\text{H--e--l--l--o}" \ #\-)
=> "H-e-l-l-o"
\]

\[
(\text{collapse-repeated-chars } "\text{H--e--l--l----o}" \ #\< \ 2)
=> "H-e--l--l--o"
\]

Returns a procedure that will split a string into lines according to the given parameters.

#:line-width
This is the target length used when deciding where to wrap lines. Default is 80.

#:expand-tabs?
Boolean describing whether tabs in the input should be expanded. Default is #t.

#:tab-width
If tabs are expanded, this will be the number of spaces to which they expand. Default is 8.

#:collapse-whitespace?
Boolean describing whether the whitespace inside the existing text should be removed or not. Default is #t.

If text is already well-formatted, and is just being wrapped to fit in a different width, then set this to '#f'. This way, many common text conventions (such as two spaces between sentences) can be preserved if in the original text. If the input text spacing cannot be trusted, then leave this setting at the default, and all repeated whitespace will be collapsed down to a single space.

#:initial-indent
Defines a string that will be put in front of the first line of wrapped text. Default is the empty string, "".

#:subsequent-indent
Defines a string that will be put in front of all lines of wrapped text, except the first one. Default is the empty string, "".

#:break-long-words?
If a single word is too big to fit on a line, this setting tells the wrapper what to do. Defaults to #t, which will break up long words. When set to #f, the line will be allowed, even though it is longer than the defined #:line-width.

The return value is a procedure of one argument, the input string, which returns a list of strings, where each element of the list is one line.

fill-string str . kwargs
Wraps the text given in string str according to the parameters provided in kwargs, or the default setting if they are not given. Returns a single string with the wrapped text. Valid keyword arguments are discussed in make-text-wrapper.

string->wrapped-lines str . kwargs
string->wrapped-lines str keywds .... Wraps the text given in string str according to the parameters provided in keywds, or the default setting if they are not given.
Returns a list of strings representing the formatted lines. Valid keyword arguments are discussed in `make-text-wrapper`.

### 7.21.6 (texinfo plain-text)

#### 7.21.6.1 Overview
Transformation from stexi to plain-text. Strives to re-create the output from `info`; comes pretty damn close.

#### 7.21.6.2 Usage

```scheme
stexi->plain-text tree
```

Transform `tree` into plain text. Returns a string.

### 7.21.7 (texinfo serialize)

#### 7.21.7.1 Overview
Serialization of stexi to plain texinfo.

#### 7.21.7.2 Usage

```scheme
stexi->texi tree
```

Serialize the stexi `tree` into plain texinfo.

### 7.21.8 (texinfo reflection)

#### 7.21.8.1 Overview
Routines to generare stexi documentation for objects and modules.

Note that in this context, an object is just a value associated with a location. It has nothing to do with GOOPS.

#### 7.21.8.2 Usage

```scheme
module-stexi-documentation sym-name [%docs-resolver] [#:docs-resolver]
```

Return documentation for the module named `sym-name`. The documentation will be formatted as stexi (see Section 7.21.1 [texinfo], page 749).

```scheme
script-stexi-documentation scriptpath
```

Return documentation for given script. The documentation will be taken from the script’s commentary, and will be returned in the stexi format (see Section 7.21.1 [texinfo], page 749).

```scheme
object-stexi-documentation _ [.] [#:force]
```

```scheme
package-stexi-standard-copying name version updated years copyright-holder permissions
```

Create a standard texinfo `copying` section.

`years` is a list of years (as integers) in which the modules being documented were released. All other arguments are strings.
package-stexi-standard-titlepage name version updated authors  [Function]
Create a standard GNU title page.

authors is a list of (name . email) pairs. All other arguments are strings.

Here is an example of the usage of this procedure:

(package-stexi-standard-titlepage
"Foolib"
"3.2"
"26 September 2006"
'(("Alyssa P Hacker" . "alyssa@example.com"))
'(2004 2005 2006)
"Free Software Foundation, Inc."
"Standard GPL permissions blurb goes here")

package-stexi-generic-menu name entries  [Function]
Create a menu from a generic alist of entries, the car of which should be the node
name, and the cdr the description. As an exception, an entry of #f will produce a
separator.

package-stexi-standard-menu name modules module-descriptions extra-entries  [Function]
Create a standard top node and menu, suitable for processing by makeinfo.

package-stexi-extended-menu name module-pairs script-pairs extra-entries  [Function]
Create an "extended" menu, like the standard menu but with a section for scripts.

package-stexi-standard-prologue name filename category description copying titlepage menu  [Function]
Create a standard prologue, suitable for later serialization to texinfo and .info creation
with makeinfo.

Returns a list of stexinfo forms suitable for passing to package-stexi-
documentation as the prologue. See [texinfo reflection package-stexi-documentation],
page 756, [texinfo reflection package-stexi-standard-titlepage], page 756, [texinfo
reflection package-stexi-standard-copying], page 755, and [texinfo reflection
package-stexi-standard-menu], page 756.

package-stexi-documentation modules name filename prologue epilogue [#:module-stexi-documentation-args] [#:scripts]  [Function]
Create stexi documentation for a package, where a package is a set of modules that
is released together.

modules is expected to be a list of module names, where a module name is a list
of symbols. The stexi that is returned will be titled name and a texinfo filename of
filename.

prologue and epilogue are lists of stexi forms that will be spliced into the output
document before and after the generated modules documentation, respectively. See
[texinfo reflection package-stexi-standard-prologue], page 756, to create a conventional
GNU texinfo prologue.

module-stexi-documentation-args is an optional argument that, if given, will be added
to the argument list when module-texi-documentation is called. For example, it
might be useful to define a #:docs-resolver argument.
package-stexi-documentation-for-include modules

module-descriptions [#:module-stexi-documentation-args]

Create stexi documentation for a package, where a package is a set of modules that is released together.

modules is expected to be a list of module names, where a module name is a list of symbols. Returns an stexinfo fragment.

Unlike package-stexi-documentation, this function simply produces a menu and the module documentations instead of producing a full texinfo document. This can be useful if you write part of your manual by hand, and just use @include to pull in the automatically generated parts.

module-stexi-documentation-args is an optional argument that, if given, will be added to the argument list when module-texi-documentation is called. For example, it might be useful to define a #:docs-resolver argument.
8 GOOPS

GOOPS is the object oriented extension to Guile. Its implementation is derived from STk-3.99.3 by Erick Gallesio and version 1.3 of Gregor Kiczales’ Tiny-Clos. It is very close in spirit to CLOS, the Common Lisp Object System, but is adapted for the Scheme language.

GOOPS is a full object oriented system, with classes, objects, multiple inheritance, and generic functions with multi-method dispatch. Furthermore its implementation relies on a meta object protocol — which means that GOOPS’s core operations are themselves defined as methods on relevant classes, and can be customised by overriding or redefining those methods.

To start using GOOPS you first need to import the (oop goops) module. You can do this at the Guile REPL by evaluating:

```
(use-modules (oop goops))
```

8.1 Copyright Notice

The material in this chapter is partly derived from the STk Reference Manual written by Erick Gallesio, whose copyright notice is as follows.

Copyright © 1993-1999 Erick Gallesio - I3S-CNRS/ESSI <eg@unice.fr> Permission to use, copy, modify, distribute, and license this software and its documentation for any purpose is hereby granted, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. This software is provided “AS IS” without express or implied warranty.

The material has been adapted for use in Guile, with the author’s permission.

8.2 Class Definition

A new class is defined with the define-class syntax:

```
(define-class class (superclass ...) slot-description ... class-option ...)
```

class is the class being defined. The list of superclasses specifies which existing classes, if any, to inherit slots and properties from. Slots hold per-instance¹ data, for instances of that class — like “fields” or “member variables” in other object oriented systems. Each slot-description gives the name of a slot and optionally some “properties” of this slot; for example its initial value, the name of a function which will access its value, and so on. Class options, slot descriptions and inheritance are discussed more below.

```
define-class name (super ...) slot-definition ... class-option ...  [syntax]
```

Define a class called name that inherits from supers, with direct slots defined by slot-definitions and class-options. The newly created class is bound to the variable name name in the current environment.

¹ Usually — but see also the #:allocation slot option.
Each *slot-definition* is either a symbol that names the slot or a list,

\[(\text{slot-name-symbol . slot-options})\]

where *slot-name-symbol* is a symbol and *slot-options* is a list with an even number of elements. The even-numbered elements of *slot-options* (counting from zero) are slot option keywords; the odd-numbered elements are the corresponding values for those keywords.

Each *class-option* is an option keyword and corresponding value.

As an example, let us define a type for representing a complex number in terms of two real numbers.\(^2\) This can be done with the following class definition:

\[
\text{(define-class <my-complex> (<number>)
  \hspace{1em} r i)}
\]

This binds the variable `<my-complex>` to a new class whose instances will contain two slots. These slots are called `r` and `i` and will hold the real and imaginary parts of a complex number. Note that this class inherits from `<number>`, which is a predefined class.\(^3\)

Slot options are described in the next section. The possible class options are as follows.

**#:metaclass** `metaclass`  
The `#:metaclass` class option specifies the metaclass of the class being defined. `metaclass` must be a class that inherits from `<class>`. For the use of metaclasses, see Section 8.11.1 [Metaobjects and the Metaobject Protocol], page 783, and Section 8.11.2 [Metaclasses], page 785.

If the `#:metaclass` option is absent, GOOPS reuses or constructs a metaclass for the new class by calling `ensure-metaclass` (see Section 8.11.5 [ensure-metaclass], page 787).

**#:name** `name`  
The `#:name` class option specifies the new class’s name. This name is used to identify the class whenever related objects - the class itself, its instances and its subclasses - are printed.

If the `#:name` option is absent, GOOPS uses the first argument to `define-class` as the class name.

### 8.3 Instance Creation and Slot Access

An instance (or object) of a defined class can be created with `make`. `make` takes one mandatory parameter, which is the class of the instance to create, and a list of optional arguments that will be used to initialize the slots of the new instance. For instance the following form

\[
\text{(define c (make <my-complex>))}
\]

creates a new `<my-complex>` object and binds it to the Scheme variable `c`.

---

\(^2\) Of course Guile already provides complex numbers, and `<complex>` is in fact a predefined class in GOOPS; but the definition here is still useful as an example.

\(^3\) `<number>` is the direct superclass of the predefined class `<complex>`; `<complex>` is the superclass of `<real>`, and `<real>` is the superclass of `<integer>`.
make (class <class>) initarg ...  
Create and return a new instance of class class, initialized using initarg . . . .
In theory, initarg . . . can have any structure that is understood by whatever methods get applied when the initialize generic function is applied to the newly allocated instance.
In practice, specialized initialize methods would normally call (next-method), and so eventually the standard GOOPS initialize methods are applied. These methods expect initargs to be a list with an even number of elements, where even-numbered elements (counting from zero) are keywords and odd-numbered elements are the corresponding values.
GOOPS processes initialization argument keywords automatically for slots whose definition includes the #:init-keyword option (see Section 8.4 [init-keyword], page 761). Other keyword value pairs can only be processed by an initialize method that is specialized for the new instance’s class. Any unprocessed keyword value pairs are ignored.

make-instance (class <class>) initarg ...
make-instance is an alias for make.

The slots of the new complex number can be accessed using slot-ref and slot-set!. slot-set! sets the value of an object slot and slot-ref retrieves it.

(slot-set! c 'r 10)
(slot-set! c 'i 3)
(slot-ref c 'r) ⇒ 10
(slot-ref c 'i) ⇒ 3

The (oop goops describe) module provides a describe function that is useful for seeing all the slots of an object; it prints the slots and their values to standard output.

(describe c)
⇒
#<my-complex> 401d8638> is an instance of class <my-complex>
Slots are:
  r = 10
  i = 3

8.4 Slot Options
When specifying a slot (in a (define-class ...) form), various options can be specified in addition to the slot’s name. Each option is specified by a keyword. The list of possible keywords is as follows.

#:init-value init-value  [slot option]
#:init-form init-form    [slot option]
#:init-thunk init-thunk  [slot option]
#:init-keyword init-keyword [slot option]

These options provide various ways to specify how to initialize the slot’s value at instance creation time.
init-value specifies a fixed initial slot value (shared across all new instances of the class).

init-thunk specifies a thunk that will provide a default value for the slot. The thunk is called when a new instance is created and should return the desired initial slot value.

init-form specifies a form that, when evaluated, will return an initial value for the slot. The form is evaluated each time that an instance of the class is created, in the lexical environment of the containing define-class expression.

init-keyword specifies a keyword that can be used to pass an initial slot value to make when creating a new instance.

Note that, since an init-value value is shared across all instances of a class, you should only use it when the initial value is an immutable value, like a constant. If you want to initialize a slot with a fresh, independently mutable value, you should use init-thunk or init-form instead. Consider the following example.

```scheme
(define-class <chbouib> ()
  (hashtab #:init-value (make-hash-table)))
```

Here only one hash table is created and all instances of <chbouib> have their hashtab slot refer to it. In order to have each instance of <chbouib> refer to a new hash table, you should instead write:

```scheme
(define-class <chbouib> ()
  (hashtab #:init-thunk make-hash-table))
```
or:

```scheme
(define-class <chbouib> ()
  (hashtab #:init-form (make-hash-table)))
```

If more than one of these options is specified for the same slot, the order of precedence, highest first is

- #:init-keyword, if init-keyword is present in the options passed to make
- #:init-thunk, #:init-form or #:init-value.

If the slot definition contains more than one initialization option of the same precedence, the later ones are ignored. If a slot is not initialized at all, its value is unbound.

In general, slots that are shared between more than one instance are only initialized at new instance creation time if the slot value is unbound at that time. However, if the new instance creation specifies a valid init keyword and value for a shared slot, the slot is re-initialized regardless of its previous value.

Note, however, that the power of GOOPS’ metaobject protocol means that everything written here may be customized or overridden for particular classes! The slot initializations described here are performed by the least specialized method of the generic function initialize, whose signature is

```scheme
(define-method (initialize (object <object>) initargs) ...)
```

The initialization of instances of any given class can be customized by defining a initialize method that is specialized for that class, and the author of the specialized method may decide to call next-method - which will result in a call to the next less specialized initialize method - at any point within the specialized code, or maybe not at all. In general, therefore, the initialization mechanisms described here may be
modified or overridden by more specialized code, or may not be supported at all for particular classes.

#:getter getter [slot option]
#:setter setter [slot option]
#:accessor accessor [slot option]

Given an object \texttt{obj} with slots named \texttt{foo} and \texttt{bar}, it is always possible to read and write those slots by calling \texttt{slot-ref} and \texttt{slot-set!} with the relevant slot name; for example:

\begin{verbatim}
(slot-ref obj 'foo)
(slot-set! obj 'bar 25)
\end{verbatim}

The #:getter, #:setter and #:accessor options, if present, tell GOOPS to create generic function and method definitions that can be used to get and set the slot value more conveniently. \texttt{getter} specifies a generic function to which GOOPS will add a method for getting the slot value. \texttt{setter} specifies a generic function to which GOOPS will add a method for setting the slot value. \texttt{accessor} specifies an accessor to which GOOPS will add methods for both getting and setting the slot value.

So if a class includes a slot definition like this:

\begin{verbatim}
(c #:getter get-count #:setter set-count #:accessor count)
\end{verbatim}

GOOPS defines generic function methods such that the slot value can be referenced using either the getter or the accessor -

\begin{verbatim}
(let ((current-count (get-count obj))) ...)
(let ((current-count (count obj))) ...)
\end{verbatim}

- and set using either the setter or the accessor -

\begin{verbatim}
(set-count obj (+ 1 current-count))
(set! (count obj) (+ 1 current-count))
\end{verbatim}

Note that

- with an accessor, the slot value is set using the generalized \texttt{set!} syntax
- in practice, it is unusual for a slot to use all three of these options: read-only, write-only and read-write slots would typically use only #:getter, #:setter and #:accessor options respectively.

The binding of the specified names is done in the environment of the \texttt{define-class} expression. If the names are already bound (in that environment) to values that cannot be upgraded to generic functions, those values are overwritten when the \texttt{define-class} expression is evaluated. For more detail, see Section 8.11.9 [ensure-generic], page 793.

#:allocation allocation [slot option]

The #:allocation option tells GOOPS how to allocate storage for the slot. Possible values for allocation are

- #:instance

Indicates that GOOPS should create separate storage for this slot in each new instance of the containing class (and its subclasses). This is the default.
• #:class
  Indicates that GOOPS should create storage for this slot that is shared by all
instances of the containing class (and its subclasses). In other words, a slot in
class C with allocation #:class is shared by all instances for which (is-a? instance c). This permits defining a kind of global variable which can be accessed
only by (in)direct instances of the class which defines the slot.

• #:each-subclass
  Indicates that GOOPS should create storage for this slot that is shared by all
direct instances of the containing class, and that whenever a subclass of the
containing class is defined, GOOPS should create a new storage for the slot that
is shared by all direct instances of the subclass. In other words, a slot with
allocation #:each-subclass is shared by all instances with the same class-of.

• #:virtual
  Indicates that GOOPS should not allocate storage for this slot. The slot defini-
tion must also include the #:slot-ref and #:slot-set! options to specify how
to reference and set the value for this slot. See the example below.

Slot allocation options are processed when defining a new class by the generic function
compute-get-n-set, which is specialized by the class’s metaclass. Hence new types
of slot allocation can be implemented by defining a new metaclass and a method for
compute-get-n-set that is specialized for the new metaclass. For an example of how
to do this, see Section 8.11.6 [Customizing Class Definition], page 790.

#:slot-ref getter [slot option]
#:slot-set! setter [slot option]

The #:slot-ref and #:slot-set! options must be specified if the slot allocation is
#:virtual, and are ignored otherwise.

gether should be a closure taking a single instance parameter that returns the current
slot value. setter should be a closure taking two parameters - instance and new-val -
that sets the slot value to new-val.

8.5 Illustrating Slot Description

To illustrate slot description, we can redefine the <my-complex> class seen before. A defi-
nition could be:

(define-class <my-complex> (<number>)
  (r #:init-value 0 #:getter get-r #:setter set-r! #:init-keyword #:r)
  (i #:init-value 0 #:getter get-i #:setter set-i! #:init-keyword #:i))

With this definition, the r and i slots are set to 0 by default, and can be initialised to other
values by calling make with the #:r and #:i keywords. Also the generic functions get-r,
set-r!, get-i and set-i! are automatically defined to read and write the slots.

(define c1 (make <my-complex> #:r 1 #:i 2))
(get-r c1) ⇒ 1
(set-r! c1 12)
(get-r c1) ⇒ 12
(define c2 (make <my-complex> #:r 2))
Accessors can both read and write a slot. So, another definition of the `<my-complex>` class, using the `#:accessor` option, could be:

```lisp
(define-class <my-complex> (<number>)
  (r #:init-value 0 #:accessor real-part #:init-keyword #:r)
  (i #:init-value 0 #:accessor imag-part #:init-keyword #:i))
```

With this definition, the `r` slot can be read with:

```lisp
(real-part c)
```

and set with:

```lisp
(set! (real-part c) new-value)
```

Suppose now that we want to manipulate complex numbers with both rectangular and polar coordinates. One solution could be to have a definition of complex numbers which uses one particular representation and some conversion functions to pass from one representation to the other. A better solution is to use virtual slots, like this:

```lisp
(define-class <my-complex> (<number>)
  ;; True slots use rectangular coordinates
  (r #:init-value 0 #:accessor real-part #:init-keyword #:r)
  (i #:init-value 0 #:accessor imag-part #:init-keyword #:i)
  ;; Virtual slots access do the conversion
  (m #:accessor magnitude #:init-keyword #:magn
     #:allocation #:virtual
     #:slot-ref (lambda (o)
                  (let ((r (slot-ref o 'r)) (i (slot-ref o 'i)))
                    (sqrt (+ (* r r) (* i i))))))
  #:slot-set! (lambda (o m)
               (let ((a (slot-ref o 'a))
                     (m (slot-ref o 'm))
                     (r (* m (cos a)))
                     (i (* m (sin a)))))
               (slot-set! o 'r r)
               (slot-set! o 'i i)))
  (a #:accessor angle #:init-keyword #:angle
     #:allocation #:virtual
     #:slot-ref (lambda (o)
                 (atan (slot-ref o 'i) (slot-ref o 'r))))
  #:slot-set! (lambda (o a)
               (let ((m (slot-ref o 'm))
                     (r (* m (cos a)))
                     (i (* m (sin a))))
                 (slot-set! o 'r r)
                 (slot-set! o 'i i)))
```

In this class definition, the magnitude `m` and angle `a` slots are virtual, and are calculated, when referenced, from the normal (i.e. `#:allocation #:instance`) slots `r` and `i`, by calling the function defined in the relevant `#:slot-ref` option. Correspondingly, writing `m` or `a` leads to calling the function defined in the `#:slot-set!` option. Thus the following expression

```lisp
(slot-set! c 'a 3)
```
permits to set the angle of the complex number.

```scheme
(define c (make <my-complex> #:r 12 #:i 20))
(real-part c) ⇒ 12
(angle c) ⇒ 1.03037682652431
(slot-set! c 'i 10)
(set! (real-part c) 1)
(describe c)

#:<<my-complex> 401e9b58> is an instance of class <my-complex>
Slots are:
   r = 1
   i = 10
   m = 10.0498756211209
   a = 1.47112767430373

Since initialization keywords have been defined for the four slots, we can now define the standard Scheme primitives `make-rectangular` and `make-polar`.

```scheme
(define make-rectangular
  (lambda (x y) (make <my-complex> #:r x #:i y)))

(define make-polar
  (lambda (x y) (make <my-complex> #:magn x #:angle y)))
```

### 8.6 Methods and Generic Functions

A GOOPS method is like a Scheme procedure except that it is specialized for a particular set of argument classes, and will only be used when the actual arguments in a call match the classes in the method definition.

```scheme
(define-method (+ (x <string>) (y <string>))
  (string-append x y))

(+ "abc" "de") ⇒ "abcde"
```

A method is not formally associated with any single class (as it is in many other object oriented languages), because a method can be specialized for a combination of several classes. If you’ve studied object orientation in non-Lispy languages, you may remember discussions such as whether a method to stretch a graphical image around a surface should be a method of the image class, with a surface as a parameter, or a method of the surface class, with an image as a parameter. In GOOPS you’d just write

```scheme
(define-method (stretch (im <image>) (sf <surface>)) ...
```

and the question of which class the method is more associated with does not need answering.

There can simultaneously be several methods with the same name but different sets of specializing argument classes; for example:

```scheme
(define-method (+ (x <string>) (y <string)) ...) 
(define-method (+ (x <matrix>) (y <matrix>)) ...) 
(define-method (+ (f <fish>) (b <bicycle>)) ...) 
```
(define-method (+ (a <foo>) (b <bar>) (c <baz>)) ...)
A generic function is a container for the set of such methods that a program intends to use.

If you look at a program’s source code, and see (+ x y) somewhere in it, conceptually what is happening is that the program at that point calls a generic function (in this case, the generic function bound to the identifier +). When that happens, Guile works out which of the generic function’s methods is the most appropriate for the arguments that the function is being called with; then it evaluates the method’s code with the arguments as formal parameters. This happens every time that a generic function call is evaluated — it isn’t assumed that a given source code call will end up invoking the same method every time.

Defining an identifier as a generic function is done with the define-generic macro. Definition of a new method is done with the define-method macro. Note that define-method automatically does a define-generic if the identifier concerned is not already a generic function, so often an explicit define-generic call is not needed.

(define-generic symbol)
Create a generic function with name symbol and bind it to the variable symbol. If symbol was previously bound to a Scheme procedure (or procedure-with-setter), the old procedure (and setter) is incorporated into the new generic function as its default procedure (and setter). Any other previous value, including an existing generic function, is discarded and replaced by a new, empty generic function.

(define-method (generic parameter ...) body ...)
Define a method for the generic function or accessor generic with parameters parameters and body ....

generic is a generic function. If generic is a variable which is not yet bound to a generic function object, the expansion of define-method will include a call to define-generic. If generic is (setter generic-with-setter), where generic-with-setter is a variable which is not yet bound to a generic-with-setter object, the expansion will include a call to define-accessor.

Each parameter must be either a symbol or a two-element list (symbol class). The symbols refer to variables in the body forms that will be bound to the parameters supplied by the caller when calling this method. The classes, if present, specify the possible combinations of parameters to which this method can be applied.

body ... are the bodies of the method definition.

define-method expressions look a little like Scheme procedure definitions of the form
(define (name formals ...) . body)

The important difference is that each formal parameter, apart from the possible “rest” argument, can be qualified by a class name: formal becomes (formal class). The meaning of this qualification is that the method being defined will only be applicable in a particular generic function invocation if the corresponding argument is an instance of class (or one of its subclasses). If more than one of the formal parameters is qualified in this way, then the method will only be applicable if each of the corresponding arguments is an instance of its respective qualifying class.

Note that unqualified formal parameters act as though they are qualified by the class <top>, which GOOPS uses to mean the superclass of all valid Scheme types, including both primitive types and GOOPS classes.
For example, if a generic function method is defined with parameters (s1 <square>) and (n <number>), that method is only applicable to invocations of its generic function that have two parameters where the first parameter is an instance of the <square> class and the second parameter is a number.

8.6.1 Accessors

An accessor is a generic function that can also be used with the generalized set! syntax (see Section 6.9.8 [Procedures with Setters], page 259). Guile will handle a call like

(set! (accessor args...) value)

by calling the most specialized method of accessor that matches the classes of args and value. define-accessor is used to bind an identifier to an accessor.

define-accessor symbol

Create an accessor with name symbol and bind it to the variable symbol. If symbol was previously bound to a Scheme procedure (or procedure-with-setter), the old procedure (and setter) is incorporated into the new accessor as its default procedure (and setter). Any other previous value, including an existing generic function or accessor, is discarded and replaced by a new, empty accessor.

8.6.2 Extending Primitives

Many of Guile's primitive procedures can be extended by giving them a generic function definition that operates in conjunction with their normal C-coded implementation. When a primitive is extended in this way, it behaves like a generic function with the C-coded implementation as its default method.

This extension happens automatically if a method is defined (by a define-method call) for a variable whose current value is a primitive. But it can also be forced by calling enable-primitive-generic!.

enable-primitive-generic! primitive

Force the creation of a generic function definition for primitive.

Once the generic function definition for a primitive has been created, it can be retrieved using primitive-generic-generic.

primitive-generic-generic primitive

Return the generic function definition of primitive.

primitive-generic-generic raises an error if primitive is not a primitive with generic capability.

8.6.3 Merging Generics

GOOPS generic functions and accessors often have short, generic names. For example, if a vector package provides an accessor for the X coordinate of a vector, that accessor may just be called x. It doesn’t need to be called, for example, vector:x, because GOOPS will work out, when it sees code like (x obj), that the vector-specific method of x should be called if obj is a vector.

That raises the question, though, of what happens when different packages define a generic function with the same name. Suppose we work with a graphical package which
Chapter 8: GOOPS

needs to use two independent vector packages for 2D and 3D vectors respectively. If both packages export x, what does the code using those packages end up with?

Section 6.20.3 [duplicate binding handlers], page 409, explains how this is resolved for conflicting bindings in general. For generics, there is a special duplicates handler, `merge-generics`, which tells the module system to merge generic functions with the same name. Here is an example:

```
(define-module (math 2D-vectors)
  #:use-module (oop goops)
  #:export (x y ...))

(define-module (math 3D-vectors)
  #:use-module (oop goops)
  #:export (x y z ...))

(define-module (my-module)
  #:use-module (oop goops)
  #:use-module (math 2D-vectors)
  #:use-module (math 3D-vectors)
  #:duplicates (merge-generics))
```

The generic function x in (my-module) will now incorporate all of the methods of x from both imported modules.

To be precise, there will now be three distinct generic functions named x: x in (math 2D-vectors), x in (math 3D-vectors), and x in (my-module); and these functions share their methods in an interesting and dynamic way.

To explain, let’s call the imported generic functions (in (math 2D-vectors) and (math 3D-vectors)) the ancestors, and the merged generic function (in (my-module)), the descendant. The general rule is that for any generic function G, the applicable methods are selected from the union of the methods of G’s descendant functions, the methods of G itself and the methods of G’s ancestor functions.

Thus ancestor functions effectively share methods with their descendants, and vice versa. In the example above, x in (math 2D-vectors) will share the methods of x in (my-module) and vice versa.\(^4\) Sharing is dynamic, so adding another new method to a descendant implies adding it to that descendant’s ancestors too.

8.6.4 Next-method

When you call a generic function, with a particular set of arguments, GOOPS builds a list of all the methods that are applicable to those arguments and orders them by how closely the method definitions match the actual argument types. It then calls the method at the top of this list. If the selected method’s code wants to call on to the next method in this list, it can do so by using `next-method`.

```
(define-method (Test (a <integer>)) (cons 'integer (next-method)))
(define-method (Test (a <number>)) (cons 'number (next-method)))
(define-method (Test a) (list 'top))
```

\(^4\) But note that x in (math 2D-vectors) doesn’t share methods with x in (math 3D-vectors), so modularity is still preserved.
With these definitions,

\begin{align*}
\text{(Test 1)} & \Rightarrow (\text{integer number top}) \\
\text{(Test 1.0)} & \Rightarrow (\text{number top}) \\
\text{(Test #t)} & \Rightarrow (\text{top})
\end{align*}

\text{next-method} is always called as just (\text{next-method}). The arguments for the next method call are always implicit, and always the same as for the original method call.

If you want to call on to a method with the same name but with a different set of arguments (as you might with overloaded methods in C++, for example), you do not use \text{next-method}, but instead simply write the new call as usual:

\begin{verbatim}
(define-method (Test (a <number>) min max)
    (if (and (>= a min) (<= a max))
        (display "Number is in range\n")
        (Test a))
\end{verbatim}

\((\text{Test 2 1 10})\)
\(-\rightarrow\)
Number is in range
\(\Rightarrow\)
\((\text{integer number top})\)

(You should be careful in this case that the Test calls do not lead to an infinite recursion, but this consideration is just the same as in Scheme code in general.)

### 8.6.5 Generic Function and Method Examples

Consider the following definitions:

\begin{verbatim}
(define-generic G)
(define-method (G (a <integer>) b) 'integer)
(define-method (G (a <real>) b) 'real)
(define-method (G a b) 'top)
\end{verbatim}

The \text{define-generic} call defines \text{G} as a generic function. The three next lines define methods for \text{G}. Each method uses a sequence of \text{parameter specializers} that specify when the given method is applicable. A specializer permits to indicate the class a parameter must belong to (directly or indirectly) to be applicable. If no specializer is given, the system defaults it to \text{<top>}. Thus, the first method definition is equivalent to

\((\text{define-method (G (a <integer>) (b <top>)) 'integer})\)

Now, let’s look at some possible calls to the generic function \text{G}:

\begin{align*}
\text{(G 2 3)} & \Rightarrow \text{integer} \\
\text{(G 2 #t)} & \Rightarrow \text{integer} \\
\text{(G 1.2 'a)} & \Rightarrow \text{real} \\
\text{(G #t #f)} & \Rightarrow \text{top} \\
\text{(G 1 2 3)} & \Rightarrow \text{error (since no method exists for 3 parameters)}
\end{align*}

The methods above use only one specializer per parameter list. But in general, any or all of a method’s parameters may be specialized. Suppose we define now:

\begin{verbatim}
(define-method (G (a <integer>) (b <number>)) 'integer-number)
(define-method (G (a <integer>) (b <real>)) 'integer-real)
\end{verbatim}
(define-method (G (a <integer>)) (b <integer>)) 'integer-integer
(define-method (G a (b <number>)) 'top-number)

With these definitions:

(G 1 2) ⇒ integer-integer
(G 1 1.0) ⇒ integer-real
(G 1 #t) ⇒ integer
(G 'a 1) ⇒ top-number

As a further example we shall continue to define operations on the <my-complex> class. Suppose that we want to use it to implement complex numbers completely. For instance a definition for the addition of two complex numbers could be

(define-method (new-+ (a <my-complex>)) (b <my-complex>))
(make-rectangular (+ (real-part a) (real-part b))
 (+ (imag-part a) (imag-part b))))

To be sure that the + used in the method new-+ is the standard addition we can do:

(define-generic new-+)
(let ((+ +))
 (define-method (new-+ (a <my-complex>)) (b <my-complex>))
 (make-rectangular (+ (real-part a) (real-part b))
 (+ (imag-part a) (imag-part b)))))

The define-generic ensures here that new-+ will be defined in the global environment. Once this is done, we can add methods to the generic function new-+ which make a closure on the + symbol. A complete writing of the new-+ methods is shown in Figure 8.1.
(define-generic new-+)

(let ((+ +))
  (define-method (new-+ (a <real>) (b <real>)) (+ a b))
  (define-method (new-+ (a <real>) (b <my-complex>))
    (make-rectangular (+ a (real-part b)) (imag-part b)))
  (define-method (new-+ (a <my-complex>) (b <real>))
    (make-rectangular (+ (real-part a) b) (imag-part a)))
  (define-method (new-+ (a <my-complex>) (b <my-complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                     (+ (imag-part a) (imag-part b))))
  (define-method (new-+ (a <number>)) a)
  (define-method (new-+) 0)
  (define-method (new-+ . args)
    (new-+ (car args)
           (apply new-+ (cdr args)))))

(set! + new-+)

Figure 8.1: Extending + to handle complex numbers

We take advantage here of the fact that generic function are not obliged to have a fixed number of parameters. The four first methods implement dyadic addition. The fifth method says that the addition of a single element is this element itself. The sixth method says that using the addition with no parameter always return 0 (as is also true for the primitive +). The last method takes an arbitrary number of parameters\(^5\). This method acts as a kind of reduce: it calls the dyadic addition on the car of the list and on the result of applying it on its rest. To finish, the set! permits to redefine the + symbol to our extended addition.

To conclude our implementation (integration?) of complex numbers, we could redefine standard Scheme predicates in the following manner:

```
(define-method (complex? c <my-complex>) #t)
(define-method (complex? c) #f)

(define-method (number? n <number>) #t)
(define-method (number? n) #f)
...  
```

\(^5\) The parameter list for a define-method follows the conventions used for Scheme procedures. In particular it can use the dot notation or a symbol to denote an arbitrary number of parameters
Standard primitives in which complex numbers are involved could also be redefined in the same manner.

### 8.6.6 Handling Invocation Errors

If a generic function is invoked with a combination of parameters for which there is no applicable method, GOOPS raises an error.

\[
\textit{no-method} \quad \text{[generic]}
\]
\[
\textit{no-method} \ (gf \ <\text{generic}>) \ \text{args} \quad \text{[method]}
\]

When an application invokes a generic function, and no methods at all have been defined for that generic function, GOOPS calls the \textit{no-method} generic function. The default method calls \texttt{goops-error} with an appropriate message.

\[
\textit{no-applicable-method} \quad \text{[generic]}
\]
\[
\textit{no-applicable-method} \ (gf \ <\text{generic}>) \ \text{args} \quad \text{[method]}
\]

When an application applies a generic function to a set of arguments, and no methods have been defined for those argument types, GOOPS calls the \textit{no-applicable-method} generic function. The default method calls \texttt{goops-error} with an appropriate message.

\[
\textit{no-next-method} \quad \text{[generic]}
\]
\[
\textit{no-next-method} \ (gf \ <\text{generic}>) \ \text{args} \quad \text{[method]}
\]

When a generic function method calls \texttt{(next-method)} to invoke the next less specialized method for that generic function, and no less specialized methods have been defined for the current generic function arguments, GOOPS calls the \textit{no-next-method} generic function. The default method calls \texttt{goops-error} with an appropriate message.

### 8.7 Inheritance

Here are some class definitions to help illustrate inheritance:

\[
\begin{align*}
\text{(define-class A () a)} \\
\text{(define-class B () b)} \\
\text{(define-class C () c)} \\
\text{(define-class D (A B) d a)} \\
\text{(define-class E (A C) e c)} \\
\text{(define-class F (D E) f)}
\end{align*}
\]

\texttt{A}, \texttt{B}, \texttt{C} have a null list of superclasses. In this case, the system will replace the null list by a list which only contains \texttt{object}, the root of all the classes defined by \texttt{define-class}. \texttt{D}, \texttt{E}, \texttt{F} use multiple inheritance: each class inherits from two previously defined classes. Those class definitions define a hierarchy which is shown in Figure 8.2. In this figure, the class \texttt{<top>} is also shown; this class is the superclass of all Scheme objects. In particular, \texttt{<top>} is the superclass of all standard Scheme types.
When a class has superclasses, its set of slots is calculated by taking the union of its own slots and those of all its superclasses. Thus each instance of D will have three slots, a, b and d). The slots of a class can be discovered using the class-slots primitive. For instance,

(class-slots A) ⇒ ((a))
(class-slots E) ⇒ ((a) (e) (c))
(class-slots F) ⇒ ((e) (c) (b) (d) (a) (f))

The ordering of the returned slots is not significant.

8.7.1 Class Precedence List

What happens when a class inherits from two or more superclasses that have a slot with the same name but incompatible definitions — for example, different init values or slot allocations? We need a rule for deciding which slot definition the derived class ends up with, and this rule is provided by the class’s Class Precedence List.\(^6\)

Another problem arises when invoking a generic function, and there is more than one method that could apply to the call arguments. Here we need a way of ordering the applicable methods, so that Guile knows which method to use first, which to use next if that method calls next-method, and so on. One of the ingredients for this ordering

---

\(^6\) This section is an adaptation of material from Jeff Dalton’s (J.Dalton@ed.ac.uk) *Brief introduction to CLOS*
is determining, for each given call argument, which of the specializing classes, from each applicable method's definition, is the most specific for that argument; and here again the class precedence list helps.

If inheritance was restricted such that each class could only have one superclass — which is known as single inheritance — class ordering would be easy. The rule would be simply that a subclass is considered more specific than its superclass.

With multiple inheritance, ordering is less obvious, and we have to impose an arbitrary rule to determine precedence. Suppose we have

```scheme
(define-class X ()
  (x #:init-value 1))

(define-class Y ()
  (x #:init-value 2))

(define-class Z (X Y)
  (...))
```

Clearly the `Z` class is more specific than `X` or `Y`, for instances of `Z`. But which is more specific out of `X` and `Y` — and hence, for the definitions above, which `#:init-value` will take effect when creating an instance of `Z`? The rule in GOOPS is that the superclasses listed earlier are more specific than those listed later. Hence `X` is more specific than `Y`, and the `#:init-value` for slot `x` in instances of `Z` will be 1.

Hence there is a linear ordering for a class and all its superclasses, from most specific to least specific, and this ordering is called the Class Precedence List of the class.

In fact the rules above are not quite enough to always determine a unique order, but they give an idea of how things work. For example, for the `F` class shown in Figure 8.2, the class precedence list is

```
(f d e a c b <object> <top>)
```

In cases where there is any ambiguity (like this one), it is a bad idea for programmers to rely on exactly what the order is. If the order for some superclasses is important, it can be expressed directly in the class definition.

The precedence list of a class can be obtained by calling `class-precedence-list`. This function returns a ordered list whose first element is the most specific class. For instance:

```scheme
(class-precedence-list B) ⇒ (#<<class> B 401b97c8>
  #<<class> <object> 401e4a10>
  #<<class> <top> 4026a9d8>)
```

Or for a more immediately readable result:

```scheme
(map class-name (class-precedence-list B)) ⇒ (B <object> <top>)
```

### 8.7.2 Sorting Methods

Now, with the idea of the class precedence list, we can state precisely how the possible methods are sorted when more than one of the methods of a generic function are applicable to the call arguments.

The rules are that
• the applicable methods are sorted in order of specificity, and the most specific method is used first, then the next if that method calls `next-method`, and so on
• a method M1 is more specific than another method M2 if the first specializing class that differs, between the definitions of M1 and M2, is more specific, in M1’s definition, for the corresponding actual call argument, than the specializing class in M2’s definition
• a class C1 is more specific than another class C2, for an object of actual class C, if C1 comes before C2 in C’s class precedence list.

8.8 Introspection

Introspection, or reflection, means being able to obtain information dynamically about GOOPS objects. It is perhaps best illustrated by considering an object oriented language that does not provide any introspection, namely C++.

Nothing in C++ allows a running program to obtain answers to the following types of question:
• What are the data members of this object or class?
• What classes does this class inherit from?
• Is this method call virtual or non-virtual?
• If I invoke `Employee::adjustHoliday()`, what class contains the `adjustHoliday()` method that will be applied?

In C++, answers to such questions can only be determined by looking at the source code, if you have access to it. GOOPS, on the other hand, includes procedures that allow answers to these questions — or their GOOPS equivalents — to be obtained dynamically, at run time.

8.8.1 Classes

A GOOPS class is itself an instance of the `<class>` class, or of a subclass of `<class>`. The definition of the `<class>` class has slots that are used to describe the properties of a class, including the following.

```
class-name class [primitive procedure]
   Return the name of class `class`. This is the value of `class`’s `name` slot.

class-direct-supers class [primitive procedure]
   Return a list containing the direct superclasses of `class`. This is the value of `class`’s `direct-supers` slot.

class-direct-slots class [primitive procedure]
   Return a list containing the slot definitions of the direct slots of `class`. This is the value of `class`’s `direct-slots` slot.

class-direct-subclasses class [primitive procedure]
   Return a list containing the direct subclasses of `class`. This is the value of `class`’s `direct-subclasses` slot.

class-direct-methods class [primitive procedure]
   Return a list of all the generic function methods that use `class` as a formal parameter specializer. This is the value of `class`’s `direct-methods` slot.
```
class-precedence-list class
[primitive procedure]
Return the class precedence list for class class (see Section 8.7.1 [Class Precedence List], page 774). This is the value of class’s cpl slot.

class-slots class
[primitive procedure]
Return a list containing the slot definitions for all class’s slots, including any slots that are inherited from superclasses. This is the value of class’s slots slot.

class-subclasses class
[procedure]
Return a list of all subclasses of class.

class-methods class
[procedure]
Return a list of all methods that use class or a subclass of class as one of its formal parameter specializers.

8.8.2 Instances

class-of value
[primitive procedure]
Return the GOOPS class of any Scheme value.

instance? object
[primitive procedure]
Return #t if object is any GOOPS instance, otherwise #f.

is-a? object class
[procedure]
Return #t if object is an instance of class or one of its subclasses.

You can use the is-a? predicate to ask whether any given value belongs to a given class, or class-of to discover the class of a given value. Note that when GOOPS is loaded (by code using the (oop goops) module) built-in classes like <string>, <list> and <number> are automatically set up, corresponding to all Guile Scheme types.

(is-a? 2.3 <number>) ⇒ #t
(is-a? 2.3 <real>) ⇒ #t
(is-a? 2.3 <string>) ⇒ #f
(is-a? ’(“a” “b”) <string>) ⇒ #f
(is-a? ’(“a” “b”) <list>) ⇒ #t
(is-a? (car ’(“a” “b”)) <string>) ⇒ #t
(is-a? <string> <class>) ⇒ #t
(is-a? <class> <string>) ⇒ #f

(class-of 2.3) ⇒ #<<class> <real> 908c708>
(class-of #(1 2 3)) ⇒ #<<class> <vector> 908cd20>
(class-of <string>) ⇒ #<<class> <class> 8bd3e10>
(class-of <class>) ⇒ #<<class> <class> 8bd3e10>

8.8.3 Slots

class-slot-definition class slot-name
[procedure]
Return the slot definition for the slot named slot-name in class class. slot-name should be a symbol.
slot-definition-name slot-def
   Extract and return the slot name from slot-def.

(slot-definition-options slot-def)
   [procedure]
   Extract and return the slot options from slot-def.

(slot-definition-allocation slot-def)
   [procedure]
   Extract and return the slot allocation option from slot-def. This is the value of the
   #:allocation keyword (see Section 8.4 [allocation], page 761), or #:instance if the
   #:allocation keyword is absent.

(slot-definition-getter slot-def)
   [procedure]
   Extract and return the slot getter option from slot-def. This is the value of the
   #:getter keyword (see Section 8.4 [getter], page 761), or #f if the #:getter keyword
   is absent.

(slot-definition-setter slot-def)
   [procedure]
   Extract and return the slot setter option from slot-def. This is the value of the
   #:setter keyword (see Section 8.4 [setter], page 761), or #f if the #:setter keyword
   is absent.

(slot-definition-accessor slot-def)
   [procedure]
   Extract and return the slot accessor option from slot-def. This is the value of the
   #:accessor keyword (see Section 8.4 [accessor], page 761), or #f if the #:accessor
   keyword is absent.

(slot-definition-init-value slot-def)
   [procedure]
   Extract and return the slot init-value option from slot-def. This is the value of the
   #:init-value keyword (see Section 8.4 [init-value], page 761), or the unbound value
   if the #:init-value keyword is absent.

(slot-definition-init-form slot-def)
   [procedure]
   Extract and return the slot init-form option from slot-def. This is the value of the
   #:init-form keyword (see Section 8.4 [init-form], page 761), or the unbound value if
   the #:init-form keyword is absent.

(slot-definition-init-thunk slot-def)
   [procedure]
   Extract and return the slot init-thunk option from slot-def. This is the value of
   the #:init-thunk keyword (see Section 8.4 [init-thunk], page 761), or #f if the
   #:init-thunk keyword is absent.

(slot-definition-init-keyword slot-def)
   [procedure]
   Extract and return the slot init-keyword option from slot-def. This is the value of
   the #:init-keyword keyword (see Section 8.4 [init-keyword], page 761), or #f if the
   #:init-keyword keyword is absent.

(slot-init-function class slot-name)
   [procedure]
   Return the initialization function for the slot named slot-name in class class. slot-
   name should be a symbol.
The returned initialization function incorporates the effects of the standard #:init-thunk, #:init-form and #:init-value slot options. These initializations can be overridden by the #:init-keyword slot option or by a specialized initialize method, so, in general, the function returned by slot-init-function may be irrelevant. For a fuller discussion, see Section 8.4 [init-value], page 761.

### 8.8.4 Generic Functions

A generic function is an instance of the <generic> class, or of a subclass of <generic>. The definition of the <generic> class has slots that are used to describe the properties of a generic function.

**generic-function-name**  
Return the name of generic function gf.

**generic-function-methods**  
Return a list of the methods of generic function gf. This is the value of gf’s methods slot.

Similarly, a method is an instance of the <method> class, or of a subclass of <method>; and the definition of the <method> class has slots that are used to describe the properties of a method.

**method-generic-function**  
Return the generic function that method belongs to. This is the value of method’s generic-function slot.

**method-specializers**  
Return a list of method’s formal parameter specializers. This is the value of method’s specializers slot.

**method-procedure**  
Return the procedure that implements method. This is the value of method’s procedure slot.

**method-source**  
Return an expression that prints to show the definition of method m.

```
(define-generic cube)
(define-method (cube (n <number>))
  (* n n n))
(map method-source (generic-function-methods cube))
⇒ ((method ((n <number>)) (* n n n)))
```
8.8.5 Accessing Slots

Any slot, regardless of its allocation, can be queried, referenced and set using the following four primitive procedures.

**slot-exists? obj slot-name**  
Return `#t` if `obj` has a slot with name `slot-name`, otherwise `#f`.

**slot-bound? obj slot-name**  
Return `#t` if the slot named `slot-name` in `obj` has a value, otherwise `#f`.  
*slot-bound?* calls the generic function `slot-missing` if `obj` does not have a slot called `slot-name` (see Section 8.8.5 [Accessing Slots], page 780).

**slot-ref obj slot-name**  
Return the value of the slot named `slot-name` in `obj`.  
*slot-ref* calls the generic function `slot-missing` if `obj` does not have a slot called `slot-name` (see Section 8.8.5 [Accessing Slots], page 780).  
*slot-ref* calls the generic function `slot-unbound` if the named slot in `obj` does not have a value (see Section 8.8.5 [Accessing Slots], page 780).

**slot-set! obj slot-name value**  
Set the value of the slot named `slot-name` in `obj`.  
*slot-set!* calls the generic function `slot-missing` if `obj` does not have a slot called `slot-name` (see Section 8.8.5 [Accessing Slots], page 780).

GOOPS stores information about slots in classes. Internally, all of these procedures work by looking up the slot definition for the slot named `slot-name` in the class `(class-of obj)`, and then using the slot definition’s “getter” and “setter” closures to get and set the slot value.

The next four procedures differ from the previous ones in that they take the class as an explicit argument, rather than assuming `(class-of obj)`. Therefore they allow you to apply the “getter” and “setter” closures of a slot definition in one class to an instance of a different class.

**slot-exists-using-class? class obj slot-name**  
Return `#t` if `class` has a slot definition for a slot with name `slot-name`, otherwise `#f`.

**slot-bound-using-class? class obj slot-name**  
Return `#t` if applying `slot-ref-using-class` to the same arguments would call the generic function `slot-unbound`, otherwise `#f`.  
*slot-bound-using-class?* calls the generic function `slot-missing` if `class` does not have a slot definition for a slot called `slot-name` (see Section 8.8.5 [Accessing Slots], page 780).

**slot-ref-using-class class obj slot-name**  
Apply the “getter” closure for the slot named `slot-name` in `class` to `obj`, and return its result.
slot-ref-using-class calls the generic function slot-missing if class does not have a slot definition for a slot called slot-name (see Section 8.8.5 [Accessing Slots], page 780).

slot-ref-using-class calls the generic function slot-unbound if the application of the “getter” closure to obj returns an unbound value (see Section 8.8.5 [Accessing Slots], page 780).

slot-set-using-class! class obj slot-name value [primitive procedure]
Apply the “setter” closure for the slot named slot-name in class to obj and value.

slot-set-using-class! calls the generic function slot-missing if class does not have a slot definition for a slot called slot-name (see Section 8.8.5 [Accessing Slots], page 780).

Slots whose allocation is per-class rather than per-instance can be referenced and set without needing to specify any particular instance.

class-slot-ref class slot-name [procedure]
Return the value of the slot named slot-name in class. The named slot must have #:class or #:each-subclass allocation (see Section 8.4 [allocation], page 761).
If there is no such slot with #:class or #:each-subclass allocation, class-slot-ref calls the slot-missing generic function with arguments class and slot-name. Otherwise, if the slot value is unbound, class-slot-ref calls the slot-unbound generic function, with the same arguments.

class-slot-set! class slot-name value [procedure]
Set the value of the slot named slot-name in class to value. The named slot must have #:class or #:each-subclass allocation (see Section 8.4 [allocation], page 761).
If there is no such slot with #:class or #:each-subclass allocation, class-slot-ref calls the slot-missing generic function with arguments class and slot-name.

When a slot-ref or slot-set! call specifies a non-existent slot name, or tries to reference a slot whose value is unbound, GOOPS calls one of the following generic functions.

slot-missing [generic]

slot-missing (class <class>) slot-name [method]

slot-missing (class <class>) (object <object>) slot-name [method]

slot-missing (class <class>) (object <object>) slot-name value [method]

When an application attempts to reference or set a class or instance slot by name, and the slot name is invalid for the specified class or object, GOOPS calls the slot-missing generic function.

The default methods all call goops-error with an appropriate message.

slot-unbound [generic]

slot-unbound (object <object>) [method]

slot-unbound (class <class>) slot-name [method]

slot-unbound (class <class>) (object <object>) slot-name [method]

When an application attempts to reference a class or instance slot, and the slot’s value is unbound, GOOPS calls the slot-unbound generic function.

The default methods all call goops-error with an appropriate message.
8.9 Error Handling

The procedure `goops-error` is called to raise an appropriate error by the default methods of the following generic functions:

- `slot-missing` (see Section 8.8.5 [slot-missing], page 780)
- `slot-unbound` (see Section 8.8.5 [slot-unbound], page 780)
- `no-method` (see Section 8.6.6 [no-method], page 773)
- `no-applicable-method` (see Section 8.6.6 [no-applicable-method], page 773)
- `no-next-method` (see Section 8.6.6 [no-next-method], page 773)

If you customize these functions for particular classes or metaclasses, you may still want to use `goops-error` to signal any error conditions that you detect.

```
goops-error format-string arg . . . [procedure]
   Raise an error with key `goops-error` and error message constructed from `format-string` and `arg` . . . . Error message formatting is as done by `scm-error`.
```

8.10 GOOPS Object Miscellany

Here we cover some points about GOOPS objects that aren’t substantial enough to merit sections on their own.

Object Equality

When GOOPS is loaded, `eqv?`, `equal?` and `=` become generic functions, and you can define methods for them, specialized for your own classes, so as to control what the various kinds of equality mean for your classes.

For example, the `assoc` procedure, for looking up an entry in an alist, is specified as using `equal?` to determine when the car of an entry in the alist is the same as the key parameter that `assoc` is called with. Hence, if you had defined a new class, and wanted to use instances of that class as the keys in an alist, you could define a method for `equal?`, for your class, to control `assoc`’s lookup precisely.

Cloning Objects

```
shallow-clone
shallow-clone (self <object>) [generic]
   Return a “shallow” clone of `self`. The default method makes a shallow clone by allocating a new instance and copying slot values from `self` to the new instance. Each slot value is copied either as an immediate value or by reference.

deeple-clone
deeple-clone (self <object>) [generic]
   Return a “deep” clone of `self`. The default method makes a deep clone by allocating a new instance and copying or cloning slot values from `self` to the new instance. If a slot value is an instance (satisfies `instance?`), it is cloned by calling `deep-clone` on that value. Other slot values are copied either as immediate values or by reference.
```
Write and Display

\texttt{write object port} \hspace{1cm} [primitive generic]
\texttt{display object port} \hspace{1cm} [primitive generic]

When GOOPS is loaded, \texttt{write} and \texttt{display} become generic functions with special methods for printing

- objects - instances of the class \texttt{<object>}
- foreign objects - instances of the class \texttt{<foreign-object>}
- classes - instances of the class \texttt{<class>}
- generic functions - instances of the class \texttt{<generic>}
- methods - instances of the class \texttt{<method>}

\texttt{write} and \texttt{display} print non-GOOPS values in the same way as the Guile primitive \texttt{write} and \texttt{display} functions.

In addition to the cases mentioned, you can of course define \texttt{write} and \texttt{display} methods for your own classes, to customize how instances of those classes are printed.

8.11 The Metaobject Protocol

At this point, we've said about as much as can be said about GOOPS without having to confront the idea of the metaobject protocol. There are a couple more topics that could be discussed in isolation first — class redefinition, and changing the class of existing instances — but in practice developers using them will be advanced enough to want to understand the metaobject protocol too, and will probably be using the protocol to customize exactly what happens during these events.

So let's plunge in. GOOPS is based on a "metaobject protocol" (aka "MOP") derived from the ones used in CLOS (the Common Lisp Object System), tiny-clos (a small Scheme implementation of a subset of CLOS functionality) and STKlos.

The MOP underlies many possible GOOPS customizations — such as defining an \texttt{initialize} method to customize the initialization of instances of an application-defined class — and an understanding of the MOP makes it much easier to explain such customizations in a precise way. And at a deeper level, understanding the MOP is a key part of understanding GOOPS, and of taking full advantage of GOOPS' power, by customizing the behaviour of GOOPS itself.

8.11.1 Metaobjects and the Metaobject Protocol

The building blocks of GOOPS are classes, slot definitions, instances, generic functions and methods. A class is a grouping of inheritance relations and slot definitions. An instance is an object with slots that are allocated following the rules implied by its class's superclasses and slot definitions. A generic function is a collection of methods and rules for determining which of those methods to apply when the generic function is invoked. A method is a procedure and a set of specializers that specify the type of arguments to which the procedure is applicable.

Of these entities, GOOPS represents classes, generic functions and methods as "metaobjects". In other words, the values in a GOOPS program that describe classes, generic
functions and methods, are themselves instances (or “objects”) of special GOOPS classes that encapsulate the behaviour, respectively, of classes, generic functions, and methods.

(The other two entities are slot definitions and instances. Slot definitions are not strictly instances, but every slot definition is associated with a GOOPS class that specifies the behaviour of the slot as regards accessibility and protection from garbage collection. Instances are of course objects in the usual sense, and there is no benefit from thinking of them as metaobjects.)

The “metaobject protocol” (or “MOP”) is the specification of the generic functions which determine the behaviour of these metaobjects and the circumstances in which these generic functions are invoked.

For a concrete example of what this means, consider how GOOPS calculates the set of slots for a class that is being defined using define-class. The desired set of slots is the union of the new class’s direct slots and the slots of all its superclasses. But define-class itself does not perform this calculation. Instead, there is a method of the initialize generic function that is specialized for instances of type <class>, and it is this method that performs the slot calculation.

initialize is a generic function which GOOPS calls whenever a new instance is created, immediately after allocating memory for a new instance, in order to initialize the new instance’s slots. The sequence of steps is as follows.

- define-class uses make to make a new instance of the <class> class, passing as initialization arguments the superclasses, slot definitions and class options that were specified in the define-class form.
- make allocates memory for the new instance, and invokes the initialize generic function to initialize the new instance’s slots.
- The initialize generic function applies the method that is specialized for instances of type <class>, and this method performs the slot calculation.

In other words, rather than being hardcoded in define-class, the default behaviour of class definition is encapsulated by generic function methods that are specialized for the class <class>.

It is possible to create a new class that inherits from <class>, which is called a “metaclass”, and to write a new initialize method that is specialized for instances of the new metaclass. Then, if the define-class form includes a #:metaclass class option whose value is the new metaclass, the class that is defined by the define-class form will be an instance of the new metaclass rather than of the default <class>, and will be defined in accordance with the new initialize method. Thus the default slot calculation, as well as any other aspect of the new class’s relationship with its superclasses, can be modified or overridden.

In a similar way, the behaviour of generic functions can be modified or overridden by creating a new class that inherits from the standard generic function class <generic>, writing appropriate methods that are specialized to the new class, and creating new generic functions that are instances of the new class.

The same is true for method metaobjects. And the same basic mechanism allows the application class author to write an initialize method that is specialized to their application class, to initialize instances of that class.
Such is the power of the MOP. Note that initialize is just one of a large number of
generic functions that can be customized to modify the behaviour of application objects
and classes and of GOOPS itself. Each following section covers a particular area of GOOPS
functionality, and describes the generic functions that are relevant for customizations of that
area.

8.11.2 Metaclasses
A metaclass is the class of an object which represents a GOOPS class. Put more succinctly,
a metaclass is a class’s class.

Most GOOPS classes have the metaclass <class> and, by default, any new class that is
created using define-class has the metaclass <class>.

But what does this really mean? To find out, let’s look in more detail at what happens
when a new class is created using define-class:

```
(define-class <my-class> (<object>) . slots)
```

Guile expands this to something like:

```
(define <my-class> (class (<object>) . slots))
```

which in turn expands to:

```
(define <my-class>
  (make <class> #:dsupers (list <object>) #:slots slots))
```

As this expansion makes clear, the resulting value of <my-class> is an instance of the
class <class> with slot values specifying the superclasses and slot definitions for the class
<my-class>. (#:dsupers and #:slots are initialization keywords for the dsupers and
dslots slots of the <class> class.)

Now suppose that you want to define a new class with a metaclass other than the default
<class>. This is done by writing:

```
(define-class <my-class2> (<object>)
  slot ...
  #:metaclass <my-metaclass>)
```

and Guile expands this to something like:

```
(define <my-class2>
  (make <my-metaclass> #:dsupers (list <object>) #:slots slots))
```

In this case, the value of <my-class2> is an instance of the more specialized class
<my-metaclass>. Note that <my-metaclass> itself must previously have been defined as
a subclass of <class>. For a full discussion of when and how it is useful to define new
metaclasses, see Section 8.11.3 [MOP Specification], page 786.

Now let’s make an instance of <my-class2>:

```
(define my-object (make <my-class2> ...))
```

All of the following statements are correct expressions of the relationships between
my-object, <my-class2>, <my-metaclass> and <class>.

- my-object is an instance of the class <my-class2>.
- <my-class2> is an instance of the class <my-metaclass>.
- <my-metaclass> is an instance of the class <class>.
• The class of \texttt{my-object} is \texttt{<my-class2>}.  
• The class of \texttt{<my-class2>} is \texttt{<my-metaclass>}.  
• The class of \texttt{<my-metaclass>} is \texttt{<class>}.  

8.11.3 MOP Specification
The aim of the MOP specification in this chapter is to specify all the customizable generic function invocations that can be made by the standard GOOPS syntax, procedures and methods, and to explain the protocol for customizing such invocations.

A generic function invocation is customizable if the types of the arguments to which it is applied are not completely determined by the lexical context in which the invocation appears. For example, the \texttt{(initialize instance initargs)} invocation in the default \texttt{make-instance} method is customizable, because the type of the \texttt{instance} argument is determined by the class that was passed to \texttt{make-instance}.

(Whereas — to give a counter-example — the \texttt{(make <generic> #:name ',name)} invocation in \texttt{define-generic} is not customizable, because all of its arguments have lexically determined types.)

When using this rule to decide whether a given generic function invocation is customizable, we ignore arguments that are expected to be handled in method definitions as a single “rest” list argument.

For each customizable generic function invocation, the \textit{invocation protocol} is explained by specifying
• what, conceptually, the applied method is intended to do  
• what assumptions, if any, the caller makes about the applied method’s side effects  
• what the caller expects to get as the applied method’s return value.

8.11.4 Instance Creation Protocol
\texttt{make <class> . initargs} (method)

• \texttt{allocate-instance class initargs} (generic)  
The applied \texttt{allocate-instance} method should allocate storage for a new instance of class \texttt{class} and return the uninitialized instance.

• \texttt{initialize instance initargs} (generic)  
\texttt{instance} is the uninitialized instance returned by \texttt{allocate-instance}. The applied method should initialize the new instance in whatever sense is appropriate for its class. The method’s return value is ignored.

\texttt{make} itself is a generic function. Hence the \texttt{make} invocation itself can be customized in the case where the new instance’s metaclass is more specialized than the default \texttt{<class>}, by defining a \texttt{make} method that is specialized to that metaclass.

Normally, however, the method for classes with metaclass \texttt{<class>} will be applied. This method calls two generic functions:
• \texttt{(allocate-instance class . initargs)}  
• \texttt{(initialize instance . initargs)}
allocate-instance allocates storage for and returns the new instance, uninitialized. You might customize allocate-instance, for example, if you wanted to provide a GOOPS wrapper around some other object programming system.

To do this, you would create a specialized metaclass, which would act as the metaclass for all classes and instances from the other system. Then define an allocate-instance method, specialized to that metaclass, which calls a Guile primitive C function (or FFI code), which in turn allocates the new instance using the interface of the other object system.

In this case, for a complete system, you would also need to customize a number of other generic functions like make and initialize, so that GOOPS knows how to make classes from the other system, access instance slots, and so on.

initialize initializes the instance that is returned by allocate-instance. The standard GOOPS methods perform initializations appropriate to the instance class.

- At the least specialized level, the method for instances of type <object> performs internal GOOPS instance initialization, and initializes the instance’s slots according to the slot definitions and any slot initialization keywords that appear in initargs.
- The method for instances of type <class> calls (next-method), then performs the class initializations described in Section 8.11.5 [Class Definition Protocol], page 787.
- and so on for generic functions, methods, operator classes . . .

Similarly, you can customize the initialization of instances of any application-defined class by defining an initialize method specialized to that class.

Imagine a class whose instances’ slots need to be initialized at instance creation time by querying a database. Although it might be possible to achieve this a combination of #:init-thunk keywords and closures in the slot definitions, it may be neater to write an initialize method for the class that queries the database once and initializes all the dependent slot values according to the results.

8.11.5 Class Definition Protocol

Here is a summary diagram of the syntax, procedures and generic functions that may be involved in class definition.

define-class (syntax)

- class (syntax)
  - make-class (procedure)
  - ensure-metaclass (procedure)
  - make metaclass . . . (generic)
    - allocate-instance (generic)
    - initialize (generic)
      - compute-cpl (generic)
      - compute-std-cpl (procedure)
      - compute-slots (generic)
      - compute-get-n-set (generic)
      - compute-getter-method (generic)
Wherever a step above is marked as “generic”, it can be customized, and the detail shown below it is only “correct” insofar as it describes what the default method of that generic function does. For example, if you write an initialize method, for some metaclass, that does not call next-method and does not call compute-cpl, then compute-cpl will not be called when a class is defined with that metaclass.

A (define-class ...) form (see Section 8.2 [Class Definition], page 759) expands to an expression which

- checks that it is being evaluated only at top level
- defines any accessors that are implied by the slot-definitions
- uses class to create the new class
- checks for a previous class definition for name and, if found, handles the redefinition by invoking class-redefinition (see Section 8.12 [Redefining a Class], page 795).

class name (super ...) slot-definition ... class-option ... [syntax]

Return a newly created class that inherits from supers, with direct slots defined by slot-definitions and class-options. For the format of slot-definitions and class-options, see Section 8.2 [define-class], page 759.

class expands to an expression which

- processes the class and slot definition options to check that they are well-formed, to convert the #:init-form option to an #:init-thunk option, to supply a default environment parameter (the current top-level environment) and to evaluate all the bits that need to be evaluated
- calls make-class to create the class with the processed and evaluated parameters.

make-class supers slots class-option ... [procedure]

Return a newly created class that inherits from supers, with direct slots defined by slots and class-options. For the format of slots and class-options, see Section 8.2 [define-class], page 759, except note that for make-class, slots is a separate list of slot definitions.

make-class

- adds <object> to the supers list if supers is empty or if none of the classes in supers have <object> in their class precedence list
- defaults the #:environment, #:name and #:metaclass options, if they are not specified by options, to the current top-level environment, the unbound value, and (ensure-metaclass supers) respectively
- checks for duplicate classes in supers and duplicate slot names in slots, and signals an error if there are any duplicates
- calls make, passing the metaclass as the first parameter and all other parameters as option keywords with values.
ensure-metaclass supers env

[procedure]

Return a metaclass suitable for a class that inherits from the list of classes in supers. The returned metaclass is the union by inheritance of the metaclasses of the classes in supers.

In the simplest case, where all the supers are straightforward classes with metaclass <class>, the returned metaclass is just <class>.

For a more complex example, suppose that supers contained one class with metaclass <operator-class> and one with metaclass <foreign-object-class>. Then the returned metaclass would be a class that inherits from both <operator-class> and <foreign-object-class>.

If supers is the empty list, ensure-metaclass returns the default GOOPS metaclass <class>.

GOOPS keeps a list of the metaclasses created by ensure-metaclass, so that each required type of metaclass only has to be created once.

The env parameter is ignored.

make metaclass initarg ...

[generic]

metaclass is the metaclass of the class being defined, either taken from the #:metaclass class option or computed by ensure-metaclass. The applied method must create and return the fully initialized class metaobject for the new class definition.

The (make metaclass initarg ...) invocation is a particular case of the instance creation protocol covered in the previous section. It will create an class metaobject with metaclass metaclass. By default, this metaobject will be initialized by the initialize method that is specialized for instances of type <class>.

The initialize method for classes (signature (initialize <class> initargs)) calls the following generic functions.

• compute-cpl class (generic)

The applied method should compute and return the class precedence list for class as a list of class metaobjects. When compute-cpl is called, the following class metaobject slots have all been initialized: name, direct-supers, direct-slots, direct- subclasses (empty), direct-methods. The value returned by compute-cpl will be stored in the cpl slot.

• compute-slots class (generic)

The applied method should compute and return the slots (union of direct and inherited) for class as a list of slot definitions. When compute-slots is called, all the class metaobject slots mentioned for compute-cpl have been initialized, plus the following: cpl, redefined (#f), environment. The value returned by compute-slots will be stored in the slots slot.

• compute-get-n-set class slot-def (generic)

initialize calls compute-get-n-set for each slot computed by compute-slots. The applied method should compute and return a pair of closures that, respectively, get and set the value of the specified slot. The get closure should have arity 1 and expect a single argument that is the instance whose slot value is to be retrieved. The set closure
should have arity 2 and expect two arguments, where the first argument is the instance whose slot value is to be set and the second argument is the new value for that slot. The closures should be returned in a two element list: \((\text{list } \text{get} \ \text{set})\).

The closures returned by \text{compute-get-n-set} are stored as part of the value of the \textit{class} metaobject’s \textit{getters-n-setters} slot. Specifically, the value of this slot is a list with the same number of elements as there are slots in the class, and each element looks either like

\[(\text{slot-name-symbol init-function . index})\]

or like

\[(\text{slot-name-symbol init-function get set})\]

Where the get and set closures are replaced by \text{index}, the slot is an instance slot and \text{index} is the slot’s index in the underlying structure: GOOPS knows how to get and set the value of such slots and so does not need specially constructed get and set closures. Otherwise, \text{get} and \text{set} are the closures returned by \text{compute-get-n-set}.

The structure of the \textit{getters-n-setters} slot value is important when understanding the next customizable generic functions that \text{initialize} calls...

- \text{compute-getter-method class gns (generic)}

  \text{initialize} calls \text{compute-getter-method} for each of the class’s slots (as determined by \text{compute-slots}) that includes a \#:\text{getter} or \#:\text{accessor} slot option. \textit{gns} is the element of the \textit{class} metaobject’s \textit{getters-n-setters} slot that specifies how the slot in question is referenced and set, as described above under \text{compute-get-n-set}. The applied method should create and return a method that is specialized for instances of type \textit{class} and uses the get closure to retrieve the slot’s value. \text{initialize} uses \text{add-method!} to add the returned method to the generic function named by the slot definition’s \#:\text{getter} or \#:\text{accessor} option.

- \text{compute-setter-method class gns (generic)}

  \text{compute-setter-method} is invoked with the same arguments as \text{compute-getter-method}, for each of the class’s slots that includes a \#:\text{setter} or \#:\text{accessor} slot option. The applied method should create and return a method that is specialized for instances of type \textit{class} and uses the set closure to set the slot’s value. \text{initialize} then uses \text{add-method!} to add the returned method to the generic function named by the slot definition’s \#:\text{setter} or \#:\text{accessor} option.

8.11.6 Customizing Class Definition

If the metaclass of the new class is something more specialized than the default \textit{<class>}, then the type of \textit{class} in the calls above is more specialized than \textit{<class>}, and hence it becomes possible to define generic function methods, specialized for the new class’s metaclass, that can modify or override the default behaviour of \text{initialize}, \text{compute-cpl} or \text{compute-get-n-set}.

\text{compute-cpl} computes the class precedence list ("CPL") for the new class (see Section 8.7.1 \[Class Precedence List\], page 774), and returns it as a list of class objects. The CPL is important because it defines a superclass ordering that is used, when a generic function is invoked upon an instance of the class, to decide which of the available generic function methods is the most specific. Hence \text{compute-cpl} could be customized in order to modify the CPL ordering algorithm for all classes with a special metaclass.
The default CPL algorithm is encapsulated by the `compute-std-cpl` procedure, which is called by the default `compute-cpl` method.

```scheme
compute-std-cpl class [procedure]
Compute and return the class precedence list for class according to the algorithm described in Section 8.7.1 [Class Precedence List], page 774.
```

`compute-slots` computes and returns a list of all slot definitions for the new class. By default, this list includes the direct slot definitions from the `define-class` form, plus the slot definitions that are inherited from the new class's superclasses. The default `compute-slots` method uses the CPL computed by `compute-cpl` to calculate this union of slot definitions, with the rule that slots inherited from superclasses are shadowed by direct slots with the same name. One possible reason for customizing `compute-slots` would be to implement an alternative resolution strategy for slot name conflicts.

`compute-get-n-set` computes the low-level closures that will be used to get and set the value of a particular slot, and returns them in a list with two elements.

The closures returned depend on how storage for that slot is allocated. The standard `compute-get-n-set` method, specialized for classes of type `<class>`, handles the standard GOOPS values for the `#:allocation` slot option (see Section 8.4 [allocation], page 761). By defining a new `compute-get-n-set` method for a more specialized metaclass, it is possible to support new types of slot allocation.

Suppose you wanted to create a large number of instances of some class with a slot that should be shared between some but not all instances of that class - say every 10 instances should share the same slot storage. The following example shows how to implement and use a new type of slot allocation to do this.

```scheme
(define-class <batched-allocation-metaclass> (<class>))
(let ((batch-allocation-count 0)
    (batch-get-n-set #f))
  (define-method (compute-get-n-set
      (class <batched-allocation-metaclass>) s)
    (case (slot-definition-allocation s)
      ((#:batched)
        ;; If we've already used the same slot storage for 10 instances,
        ;; reset variables.
        (if (= batch-allocation-count 10)
            (begin
              (set! batch-allocation-count 0)
              (set! batch-get-n-set #f))
            ;; If we don't have a current pair of get and set closures,
            ;; create one. make-closure-variable returns a pair of closures
            ;; around a single Scheme variable - see goops.scm for details.
            (or batch-get-n-set
                (set! batch-get-n-set (make-closure-variable)))
            ;; Increment batch allocation count.
            (set! batch-allocation-count (+ batch-allocation-count 1))
            batch-get-n-set)
```
(define-class <class-using-batched-slot> ()
  (c #:allocation #:batched)
  ...
  #:metaclass <batched-allocation-metaclass>)

The usage of compute-getter-method and compute-setter-method is described in Section 8.11.5 [Class Definition Protocol], page 787.

compute-cpl and compute-get-n-set are called by the standard initialize method for classes whose metaclass is <class>. But initialize itself can also be modified, by defining an initialize method specialized to the new class’s metaclass. Such a method could complete override the standard behaviour, by not calling (next-method) at all, but more typically it would perform additional class initialization steps before and/or after calling (next-method) for the standard behaviour.

8.11.7 Method Definition

define-method (syntax)
  • add-method! target method (generic)

define-method invokes the add-method! generic function to handle adding the new method to a variety of possible targets. GOOPS includes methods to handle target as
  • a generic function (the most common case)
  • a procedure
  • a primitive generic (see Section 8.6.2 [Extending Primitives], page 768)

By defining further methods for add-method!, you can theoretically handle adding methods to further types of target.

8.11.8 Method Definition Internals

define-method:
  • checks the form of the first parameter, and applies the following steps to the accessor’s setter if it has the (setter ...) form
  • interpolates a call to define-generic or define-accessor if a generic function is not already defined with the supplied name
  • calls method with the parameters and body, to make a new method instance
  • calls add-method! to add this method to the relevant generic function.

method (parameter ...) body ...
  [syntax]

Make a method whose specializers are defined by the classes in parameters and whose procedure definition is constructed from the parameter symbols and body forms.

The parameter and body parameters should be as for define-method (see Section 8.6 [define-method], page 766).
method:
• extracts formals and specializing classes from the parameters, defaulting the class for
  unspecialized parameters to <top>
• creates a closure using the formals and the body forms
• calls make with metaclass <method> and the specializers and closure using the
  #:specializers and #:procedure keywords.

make-method specializers procedure
  Make a method using specializers and procedure.
  specializers should be a list of classes that specifies the parameter combinations to
  which this method will be applicable.
  procedure should be the closure that will applied to the generic function parameters
  when this method is invoked.

make-method is a simple wrapper around make with metaclass <method>.

add-method! target method
  Generic function for adding method method to target.

add-method! (generic <generic>) (method <method>)
  Add method method to the generic function generic.

add-method! (proc <procedure>) (method <method>)
  If proc is a procedure with generic capability (see Section 8.6.2 [generic-capability?],
  page 768), upgrade it to a primitive generic and add method to its generic function
  definition.

add-method! (pg <primitive-generic>) (method <method>)
  Add method method to the generic function definition of pg.
  Implementation: (add-method! (primitive-generic-generic pg) method).

add-method! (whatever <top>) (method <method>)
  Raise an error indicating that whatever is not a valid generic function.

8.11.9 Generic Function Internals

define-generic calls ensure-generic to upgrade a pre-existing procedure value, or make
with metaclass <generic> to create a new generic function.

define-accessor calls ensure-accessor to upgrade a pre-existing procedure value, or
make-accessor to create a new accessor.

ensure-generic old-definition [name]
  Return a generic function with name name, if possible by using or upgrading old-
  definition. If unspecified, name defaults to #f.
  If old-definition is already a generic function, it is returned unchanged.
  If old-definition is a Scheme procedure or procedure-with-setter, ensure-generic
  returns a new generic function that uses old-definition for its default procedure and
  setter.
  Otherwise ensure-generic returns a new generic function with no defaults and no
  methods.
make-generic [name] [procedure]

Return a new generic function with name (car name). If unspecified, name defaults to #f.

ensure-generic calls make with metaclasses <generic> and <generic-with-setter>, depending on the previous value of the variable that it is trying to upgrade.

make-generic is a simple wrapper for make with metaclass <generic>.

ensure-accessor proc [name] [procedure]

Return an accessor with name name, if possible by using or upgrading proc. If unspecified, name defaults to #f.

If proc is already an accessor, it is returned unchanged.

If proc is a Scheme procedure, procedure-with-setter or generic function, ensure-accessor returns an accessor that reuses the reusable elements of proc.

Otherwise ensure-accessor returns a new accessor with no defaults and no methods.

make-accessor [name] [procedure]

Return a new accessor with name (car name). If unspecified, name defaults to #f.

ensure-accessor calls make with metaclass <generic-with-setter>, as well as calls to ensure-generic, make-accessor and (tail recursively) ensure-accessor.

make-accessor calls make twice, first with metaclass <generic> to create a generic function for the setter, then with metaclass <generic-with-setter> to create the accessor, passing the setter generic function as the value of the #:setter keyword.

8.11.10 Generic Function Invocation

There is a detailed and customizable protocol involved in the process of invoking a generic function — i.e., in the process of deciding which of the generic function’s methods are applicable to the current arguments, and which one of those to apply. Here is a summary diagram of the generic functions involved.

apply-generic (generic)

• no-method (generic)
• compute-applicable-methods (generic)
• sort-applicable-methods (generic)
  • method-more-specific? (generic)
• apply-methods (generic)
  • apply-method (generic)
  • no-next-method (generic)
• no-applicable-method

We do not yet have full documentation for these. Please refer to the code (oop/goops.scm) for details.
8.12 Redefining a Class

Suppose that a class `<my-class>` is defined using `define-class` (see Section 8.2 [define-class], page 759), with slots that have accessor functions, and that an application has created several instances of `<my-class>` using `make` (see Section 8.3 [make], page 760). What then happens if `<my-class>` is redefined by calling `define-class` again?

8.12.1 Default Class Redefinition Behaviour

GOOPS’ default answer to this question is as follows.

- All existing direct instances of `<my-class>` are converted to be instances of the new class. This is achieved by preserving the values of slots that exist in both the old and new definitions, and initializing the values of new slots in the usual way (see Section 8.3 [make], page 760).
- All existing subclasses of `<my-class>` are redefined, as though the `define-class` expressions that defined them were re-evaluated following the redefinition of `<my-class>`, and the class redefinition process described here is applied recursively to the redefined subclasses.
- Once all of its instances and subclasses have been updated, the class metaobject previously bound to the variable `<my-class>` is no longer needed and so can be allowed to be garbage collected.

To keep things tidy, GOOPS also needs to do a little housekeeping on methods that are associated with the redefined class.

- Slot accessor methods for slots in the old definition should be removed from their generic functions. They will be replaced by accessor methods for the slots of the new class definition.
- Any generic function method that uses the old `<my-class>` metaobject as one of its formal parameter specializers must be updated to refer to the new `<my-class>` metaobject. (Whenever a new generic function method is defined, `define-method` adds the method to a list stored in the class metaobject for each class used as a formal parameter specializer, so it is easy to identify all the methods that must be updated when a class is redefined.)

If this class redefinition strategy strikes you as rather counter-intuitive, bear in mind that it is derived from similar behaviour in other object systems such as CLOS, and that experience in those systems has shown it to be very useful in practice.

Also bear in mind that, like most of GOOPS’ default behaviour, it can be customized...

8.12.2 Customizing Class Redefinition

When `define-class` notices that a class is being redefined, it constructs the new class metaobject as usual, then invokes the `class-redefinition` generic function with the old and new classes as arguments. Therefore, if the old or new classes have metaclasses other than the default `<class>`, class redefinition behaviour can be customized by defining a `class-redefinition` method that is specialized for the relevant metaclasses.
class-redefinition

Handle the class redefinition from old-class to new-class, and return the new class
metaobject that should be bound to the variable specified by define-class’s first
argument.

class-redefinition (old-class <class>) (new-class <class>)

Implements GOOPS’ default class redefinition behaviour, as described in
Section 8.12.1 [Default Class Redefinition Behaviour], page 795. Returns the
metaobject for the new class definition.

The default class-redefinition method, for classes with the default metaclass
<class>, calls the following generic functions, which could of course be individually
customized.

remove-class-accessors! old

The default remove-class-accessors! method removes the accessor methods of the
old class from all classes which they specialize.

update-direct-method! method old new

The default update-direct-method! method substitutes the new class for the old in
all methods specialized to the old class.

update-direct-subclass! subclass old new

The default update-direct-subclass! method invokes class-redefinition recur-
sively to handle the redefinition of subclasses.

An alternative class redefinition strategy could be to leave all existing instances as in-
stances of the old class, but accepting that the old class is now “nameless”, since its name
has been taken over by the new definition. In this strategy, any existing subclasses could
also be left as they are, on the understanding that they inherit from a nameless superclass.

This strategy is easily implemented in GOOPS, by defining a new metaclass, that will be
used as the metaclass for all classes to which the strategy should apply, and then defining
a class-redefinition method that is specialized for this metaclass:

(define-class <can-be-nameless> (<class>))

(define-method (class-redefinition (old <can-be-nameless>)
                 (new <class>))

new)

When customization can be as easy as this, aren’t you glad that GOOPS implements
the far more difficult strategy as its default!

8.13 Changing the Class of an Instance

When a class is redefined, any existing instance of the redefined class will be modified for
the new class definition before the next time that any of the instance’s slots is referenced
or set. GOOPS modifies each instance by calling the generic function change-class.

More generally, you can change the class of an existing instance at any time by invoking
the generic function change-class with two arguments: the instance and the new class.
The default method for \texttt{change-class} decides how to implement the change of class by looking at the slot definitions for the instance’s existing class and for the new class. If the new class has slots with the same name as slots in the existing class, the values for those slots are preserved. Slots that are present only in the existing class are discarded. Slots that are present only in the new class are initialized using the corresponding slot definition’s init function (see Section 8.8.1 [slot-init-function], page 776).

\begin{verbatim}
change-class instance new-class [generic]
change-class (obj <object>) (new <class>) [method]
  Modify instance \texttt{obj} to make it an instance of class \texttt{new}.
  The value of each of \texttt{obj}’s slots is preserved only if a similarly named slot exists in \texttt{new}; any other slot values are discarded.
  The slots in \texttt{new} that do not correspond to any of \texttt{obj}’s pre-existing slots are initialized according to \texttt{new}’s slot definitions’ init functions.
\end{verbatim}

The default \texttt{change-class} method also invokes another generic function, \texttt{update-instance-for-different-class}, as the last thing that it does before returning. The applied \texttt{update-instance-for-different-class} method can make any further adjustments to \texttt{new-instance} that are required to complete or modify the change of class. The return value from the applied method is ignored.

\begin{verbatim}
update-instance-for-different-class old-instance new-instance [generic]
  A generic function that can be customized to put finishing touches to an instance whose class has just been changed. The default \texttt{update-instance-for-different-class} method does nothing.
\end{verbatim}

Customized change of class behaviour can be implemented by defining \texttt{change-class} methods that are specialized either by the class of the instances to be modified or by the metaclass of the new class.
9 Guile Implementation

At some point, after one has been programming in Scheme for some time, another level of Scheme comes into view: its implementation. Knowledge of how Scheme can be implemented turns out to be necessary to become an expert hacker. As Peter Norvig notes in his retrospective on PAIP\(^1\), “The expert Lisp programmer eventually develops a good ‘efficiency model’.”

By this Norvig means that over time, the Lisp hacker eventually develops an understanding of how much her code “costs” in terms of space and time.

This chapter describes Guile as an implementation of Scheme: its history, how it represents and evaluates its data, and its compiler. This knowledge can help you to make that step from being one who is merely familiar with Scheme to being a real hacker.

9.1 A Brief History of Guile

Guile is an artifact of historical processes, both as code and as a community of hackers. It is sometimes useful to know this history when hacking the source code, to know about past decisions and future directions.

Of course, the real history of Guile is written by the hackers hacking and not the writers writing, so we round up the section with a note on current status and future directions.

9.1.1 The Emacs Thesis

The story of Guile is the story of bringing the development experience of Emacs to the mass of programs on a GNU system.

Emacs, when it was first created in its GNU form in 1984, was a new take on the problem of “how to make a program”. The Emacs thesis is that it is delightful to create composite programs based on an orthogonal kernel written in a low-level language together with a powerful, high-level extension language.

Extension languages foster extensible programs, programs which adapt readily to different users and to changing times. Proof of this can be seen in Emacs’ current and continued existence, spanning more than a quarter-century.

Besides providing for modification of a program by others, extension languages are good for intension as well. Programs built in “the Emacs way” are pleasurable and easy for their authors to flesh out with the features that they need.

After the Emacs experience was appreciated more widely, a number of hackers started to consider how to spread this experience to the rest of the GNU system. It was clear that the easiest way to Emacsify a program would be to embed a shared language implementation into it.

9.1.2 Early Days

Tom Lord was the first to fully concentrate his efforts on an embeddable language runtime, which he named “GEL”, the GNU Extension Language.

\(^1\) PAIP is the common abbreviation for Paradigms of Artificial Intelligence Programming, an old but still useful text on Lisp. Norvig’s retrospective sums up the lessons of PAIP, and can be found at http://norvig.com/Lisp-retro.html.
GEL was the product of converting SCM, Aubrey Jaffer’s implementation of Scheme, into something more appropriate to embedding as a library. (SCM was itself based on an implementation by George Carrette, SIOD.)

Lord managed to convince Richard Stallman to dub GEL the official extension language for the GNU project. It was a natural fit, given that Scheme was a cleaner, more modern Lisp than Emacs Lisp. Part of the argument was that eventually when GEL became more capable, it could gain the ability to execute other languages, especially Emacs Lisp.

Due to a naming conflict with another programming language, Jim Blandy suggested a new name for GEL: “Guile”. Besides being a recursive acronym, “Guile” craftily follows the naming of its ancestors, “Planner”, “Conniver”, and “Schemer”. (The latter was truncated to “Scheme” due to a 6-character file name limit on an old operating system.) Finally, “Guile” suggests “guy-ell”, or “Guy L. Steele”, who, together with Gerald Sussman, originally discovered Scheme.

Around the same time that Guile (then GEL) was readying itself for public release, another extension language was gaining in popularity, Tcl. Many developers found advantages in Tcl because of its shell-like syntax and its well-developed graphical widgets library, Tk. Also, at the time there was a large marketing push promoting Tcl as a “universal extension language”.

Richard Stallman, as the primary author of GNU Emacs, had a particular vision of what extension languages should be, and Tcl did not seem to him to be as capable as Emacs Lisp. He posted a criticism to the comp.lang.tcl newsgroup, sparking one of the internet’s legendary flamewars. As part of these discussions, retrospectively dubbed the “Tcl Wars”, he announced the Free Software Foundation’s intent to promote Guile as the extension language for the GNU project.

It is a common misconception that Guile was created as a reaction to Tcl. While it is true that the public announcement of Guile happened at the same time as the “Tcl wars”, Guile was created out of a condition that existed outside the polemic. Indeed, the need for a powerful language to bridge the gap between extension of existing applications and a more fully dynamic programming environment is still with us today.

9.1.3 A Scheme of Many Maintainers

Surveying the field, it seems that Scheme implementations correspond with their maintainers on an N-to-1 relationship. That is to say, that those people that implement Schemes might do so on a number of occasions, but that the lifetime of a given Scheme is tied to the maintainership of one individual.

Guile is atypical in this regard.

Tom Lord maintained Guile for its first year and a half or so, corresponding to the end of 1994 through the middle of 1996. The releases made in this time constitute an arc from SCM as a standalone program to Guile as a reusable, embeddable library, but passing through a explosion of features: embedded Tcl and Tk, a toolchain for compiling and disassembling Java, addition of a C-like syntax, creation of a module system, and a start at a rich POSIX interface.

Only some of those features remain in Guile. There were ongoing tensions between providing a small, embeddable language, and one which had all of the features (e.g. a graphical toolkit) that a modern Emacs might need. In the end, as Guile gained in uptake,
the development team decided to focus on depth, documentation and orthogonality rather than on breadth. This has been the focus of Guile ever since, although there is a wide range of third-party libraries for Guile.

Jim Blandy presided over that period of stabilization, in the three years until the end of 1999, when he too moved on to other projects. Since then, Guile has had a group maintainership. The first group was Maciej Stachowiak, Mikael Djurfeldt, and Marius Vollmer, with Vollmer staying on the longest. By late 2007, Vollmer had mostly moved on to other things, so Neil Jerram and Ludovic Courtès stepped up to take on the primary maintenance responsibility. Jerram and Courtès were joined by Andy Wingo in late 2009.

Of course, a large part of the actual work on Guile has come from other contributors too numerous to mention, but without whom the world would be a poorer place.

9.1.4 A Timeline of Selected Guile Releases

guile-i — 4 February 1995
SCM, turned into a library.

guile-ii — 6 April 1995
A low-level module system was added. Tcl/Tk support was added, allowing extension of Scheme by Tcl or vice versa. POSIX support was improved, and there was an experimental stab at Java integration.

guile-iii — 18 August 1995
The C-like syntax, ctax, was improved, but mostly this release featured a start at the task of breaking Guile into pieces.

1.0 — 5 January 1997

#f was distinguished from '('). User-level, cooperative multi-threading was added. Source-level debugging became more useful, and programmer's and user's manuals were begun. The module system gained a high-level interface, which is still used today in more or less the same form.

1.1 — 16 May 1997
1.2 — 24 June 1997
Support for Tcl/Tk and ctax were split off as separate packages, and have remained there since. Guile became more compatible with SCSH, and more useful as a UNIX scripting language. Libguile could now be built as a shared library, and third-party extensions written in C became loadable via dynamic linking.

1.3.0 — 19 October 1998
Command-line editing became much more pleasant through the use of the readline library. The initial support for internationalization via multi-byte strings was removed; 10 years were to pass before proper internationalization would land again. Initial Emacs Lisp support landed, ports gained better support for file descriptors, and fluids were added.
A long list of lispy features were added: hooks, Common Lisp’s `format`, optional and keyword procedure arguments, `getopt-long`, sorting, random numbers, and many other fixes and enhancements. Guile also gained an interactive debugger, interactive help, and better backtraces.

Guile gained support for the R5RS standard, and added a number of SRFI modules. The module system was expanded with programmatic support for identifier selection and renaming. The GOOPS object system was merged into Guile core.

Guile’s arbitrary-precision arithmetic switched to use the GMP library, and added support for exact rationals. Guile’s embedded user-space threading was removed in favor of POSIX pre-emptive threads, providing true multiprocessing. Gettext support was added, and Guile’s C API was cleaned up and orthogonized in a massive way.

A virtual machine was added to Guile, along with the associated compiler and toolchain. Support for internationalization was finally reimplemented, in terms of unicode, locales, and libunistring. Running Guile instances became controllable and debuggable from within Emacs, via Geiser. Guile caught up to features found in a number of other Schemes: SRFI-18 threads, module-hygienic macros, a profiler, tracer, and debugger, SSAX XML integration, bytevectors, a dynamic FFI, delimited continuations, module versions, and partial support for R6RS.

The virtual machine and introduced in 2.0 was completely rewritten, along with much of the compiler and toolchain. This speeds up many Guile programs as well as reducing startup time and memory usage. A PEG parser toolkit was added, making it easier to write other language frontends.

Guile has achieved much of what it set out to achieve, but there is much remaining to do. There is still the old problem of bringing existing applications into a more Emacs-like experience. Guile has had some successes in this respect, but still most applications in the GNU system are without Guile integration.

Getting Guile to those applications takes an investment, the “hacktivation energy” needed to wire Guile into a program that only pays off once it is good enough to enable new kinds of behavior. This would be a great way for new hackers to contribute: take an application that you use and that you know well, think of something that it can’t yet do, and figure out a way to integrate Guile and implement that task in Guile.

With time, perhaps this exposure can reverse itself, whereby programs can run under Guile instead of vice versa, eventually resulting in the Emacsification of the entire GNU
system. Indeed, this is the reason for the naming of the many Guile modules that live in the
ice-9 namespace, a nod to the fictional substance in Kurt Vonnegut’s novel, Cat’s Cradle,
capable of acting as a seed crystal to crystallize the mass of software.

Implicit to this whole discussion is the idea that dynamic languages are somehow better
than languages like C. While languages like C have their place, Guile’s take on this question
is that yes, Scheme is more expressive than C, and more fun to write. This realization
carries an imperative with it to write as much code in Scheme as possible rather than in
other languages.

These days it is possible to write extensible applications almost entirely from high-
level languages, through byte-code and native compilation, speed gains in the underlying
hardware, and foreign call interfaces in the high-level language. Smalltalk systems are like
this, as are Common Lisp-based systems. While there already are a number of pure-Guile
applications out there, users still need to drop down to C for some tasks: interfacing to
system libraries that don’t have prebuilt Guile interfaces, and for some tasks requiring high
performance. Native ahead-of-time compilation, planned for Guile 3.0, should help with
this.

Still, even with an all-Guile application, sometimes you want to provide an opportunity
for users to extend your program from a language with a syntax that is closer to C, or to
Python. Another interesting idea to consider is compiling e.g. Python to Guile. It’s not
that far-fetched of an idea: see for example IronPython or JRuby.

And then there’s Emacs itself. Guile’s Emacs Lisp support has reached an excellent
level of correctness, robustness, and speed. However there is still work to do to finish its
integration into Emacs itself. This will give lots of exciting things to Emacs: native threads,
a real object system, more sophisticated types, cleaner syntax, and access to all of the Guile
extensions.

Finally, there is another axis of crystallization, the axis between different Scheme imple-
mentations. Guile does not yet support the latest Scheme standard, R7RS, and should do
so. Like all standards, R7RS is imperfect, but supporting it will allow more code to run on
Guile without modification, and will allow Guile hackers to produce code compatible with
other schemes. Help in this regard would be much appreciated.

9.2 Data Representation

Scheme is a latently-typed language; this means that the system cannot, in general, deter-
mine the type of a given expression at compile time. Types only become apparent at run
time. Variables do not have fixed types; a variable may hold a pair at one point, an integer
at the next, and a thousand-element vector later. Instead, values, not variables, have fixed
types.

In order to implement standard Scheme functions like pair? and string? and provide
garbage collection, the representation of every value must contain enough information to
accurately determine its type at run time. Often, Scheme systems also use this information
to determine whether a program has attempted to apply an operation to an inappropriately
typed value (such as taking the car of a string).

Because variables, pairs, and vectors may hold values of any type, Scheme implementa-
tions use a uniform representation for values — a single type large enough to hold either a
complete value or a pointer to a complete value, along with the necessary typing information.

The following sections will present a simple typing system, and then make some refinements to correct its major weaknesses. We then conclude with a discussion of specific choices that Guile has made regarding garbage collection and data representation.

9.2.1 A Simple Representation

The simplest way to represent Scheme values in C would be to represent each value as a pointer to a structure containing a type indicator, followed by a union carrying the real value. Assuming that SCM is the name of our universal type, we can write:

```c
enum type { integer, pair, string, vector, ... };

typedef struct value *SCM;

struct value {
    enum type type;
    union {
        int integer;
        struct { SCM car, cdr; } pair;
        struct { int length; char *elts; } string;
        struct { int length; SCM *elts; } vector;
        ...
    } value;
};
```

with the ellipses replaced with code for the remaining Scheme types.

This representation is sufficient to implement all of Scheme’s semantics. If x is an SCM value:

- To test if x is an integer, we can write `x->type == integer`.
- To find its value, we can write `x->value.integer`.
- To test if x is a vector, we can write `x->type == vector`.
- If we know x is a vector, we can write `x->value.vector.elts[0]` to refer to its first element.
- If we know x is a pair, we can write `x->value.pair.car` to extract its car.

9.2.2 Faster Integers

Unfortunately, the above representation has a serious disadvantage. In order to return an integer, an expression must allocate a `struct value`, initialize it to represent that integer, and return a pointer to it. Furthermore, fetching an integer’s value requires a memory reference, which is much slower than a register reference on most processors. Since integers are extremely common, this representation is too costly, in both time and space. Integers should be very cheap to create and manipulate.

One possible solution comes from the observation that, on many architectures, heap-allocated data (i.e., what you get when you call `malloc`) must be aligned on an eight-byte boundary. (Whether or not the machine actually requires it, we can write our own allocator...
for struct value objects that assures this is true.) In this case, the lower three bits of the structure’s address are known to be zero.

This gives us the room we need to provide an improved representation for integers. We make the following rules:

- If the lower three bits of an SCM value are zero, then the SCM value is a pointer to a struct value, and everything proceeds as before.
- Otherwise, the SCM value represents an integer, whose value appears in its upper bits.

Here is C code implementing this convention:

```c
enum type { pair, string, vector, ... };

typedef struct value *SCM;

struct value {
    enum type type;
    union {
        struct { SCM car, cdr; } pair;
        struct { int length; char *elts; } string;
        struct { int length; SCM *elts; } vector;
        ...
    } value;
};
```

```c
#define POINTER_P(x) (((int) (x) & 7) == 0)
#define INTEGER_P(x) (! POINTER_P (x))
#define GET_INTEGER(x) ((int) (x) >> 3)
#define MAKE_INTEGER(x) ((SCM) (((x) << 3) | 1))
```

Notice that integer no longer appears as an element of enum type, and the union has lost its integer member. Instead, we use the POINTER_P and INTEGER_P macros to make a coarse classification of values into integers and non-integers, and do further type testing as before.

Here’s how we would answer the questions posed above (again, assume x is an SCM value):

- To test if x is an integer, we can write INTEGER_P (x).
- To find its value, we can write GET_INTEGER (x).
- To test if x is a vector, we can write:
  ```c
  POINTER_P (x) && x->type == vector
  ```
  Given the new representation, we must make sure x is truly a pointer before we dereference it to determine its complete type.
- If we know x is a vector, we can write x->value.vector.elts[0] to refer to its first element, as before.
- If we know x is a pair, we can write x->value.pair.car to extract its car, just as before.
This representation allows us to operate more efficiently on integers than the first. For example, if \( x \) and \( y \) are known to be integers, we can compute their sum as follows:

\[
\text{MAKE_INTEGER (GET_INTEGER (x) + GET_INTEGER (y))}
\]

Now, integer math requires no allocation or memory references. Most real Scheme systems actually implement addition and other operations using an even more efficient algorithm, but this essay isn’t about bit-twiddling. (Hint: how do you decide when to overflow to a bignum? How would you do it in assembly?)

### 9.2.3 Cheaper Pairs

However, there is yet another issue to confront. Most Scheme heaps contain more pairs than any other type of object; Jonathan Rees said at one point that pairs occupy 45% of the heap in his Scheme implementation, Scheme 48. However, our representation above spends three SCM-sized words per pair — one for the type, and two for the \texttt{car} and \texttt{cdr}. Is there any way to represent pairs using only two words?

Let us refine the convention we established earlier. Let us assert that:

- If the bottom three bits of an SCM value are \#b000, then it is a pointer, as before.
- If the bottom three bits are \#b001, then the upper bits are an integer. This is a bit more restrictive than before.
- If the bottom two bits are \#b010, then the value, with the bottom three bits masked out, is the address of a pair.

Here is the new C code:

```c
enum type { string, vector, ... };

typedef struct value *SCM;

struct value {
    enum type type;
    union {
        struct { int length; char *elts; } string;
        struct { int length; SCM *elts; } vector;
        ...
    } value;
};

struct pair {
    SCM car, cdr;
};

#define POINTER_P(x) (((int) (x) & 7) == 0)
#define INTEGER_P(x) (((int) (x) & 7) == 1)
#define GET_INTEGER(x) ((int) (x) >> 3)
#define MAKE_INTEGER(x) ((SCM) (((x) << 3) | 1))
#define PAIR_P(x) (((int) (x) & 7) == 2)
```
#define GET_PAIR(x) ((struct pair *) ((int) (x) & ~7))

Notice that enum type and struct value now only contain provisions for vectors and strings; both integers and pairs have become special cases. The code above also assumes that an int is large enough to hold a pointer, which isn’t generally true.

Our list of examples is now as follows:

- To test if \( x \) is an integer, we can write INTEGER_P (\( x \)); this is as before.
- To find its value, we can write GET_INTEGER (\( x \)), as before.
- To test if \( x \) is a vector, we can write:
  
  \[
  \text{POINTER_P (} x \text{) && x->type == vector}
  \]
  
  We must still make sure that \( x \) is a pointer to a struct value before dereferencing it to find its type.

- If we know \( x \) is a vector, we can write \( x->value.vector.elts[0] \) to refer to its first element, as before.
- We can write PAIR_P (\( x \)) to determine if \( x \) is a pair, and then write GET_PAIR (\( x \))->car to refer to its car.

This change in representation reduces our heap size by 15%. It also makes it cheaper to decide if a value is a pair, because no memory references are necessary; it suffices to check the bottom two bits of the SCM value. This may be significant when traversing lists, a common activity in a Scheme system.

Again, most real Scheme systems use a slightly different implementation; for example, if GET_PAIR subtracts off the low bits of \( x \), instead of masking them off, the optimizer will often be able to combine that subtraction with the addition of the offset of the structure member we are referencing, making a modified pointer as fast to use as an unmodified pointer.

### 9.2.4 Conservative Garbage Collection

Aside from the latent typing, the major source of constraints on a Scheme implementation’s data representation is the garbage collector. The collector must be able to traverse every live object in the heap, to determine which objects are not live, and thus collectable.

There are many ways to implement this. Guile’s garbage collection is built on a library, the Boehm-Demers-Weiser conservative garbage collector (BDW-GC). The BDW-GC “just works”, for the most part. But since it is interesting to know how these things work, we include here a high-level description of what the BDW-GC does.

Garbage collection has two logical phases: a mark phase, in which the set of live objects is enumerated, and a sweep phase, in which objects not traversed in the mark phase are collected. Correct functioning of the collector depends on being able to traverse the entire set of live objects.

In the mark phase, the collector scans the system’s global variables and the local variables on the stack to determine which objects are immediately accessible by the C code. It then scans those objects to find the objects they point to, and so on. The collector logically sets a mark bit on each object it finds, so each object is traversed only once.

When the collector can find no unmarked objects pointed to by marked objects, it assumes that any objects that are still unmarked will never be used by the program (since
there is no path of dereferences from any global or local variable that reaches them)
and deallocates them.

In the above paragraphs, we did not specify how the garbage collector finds the global and
local variables; as usual, there are many different approaches. Frequently, the programmer
must maintain a list of pointers to all global variables that refer to the heap, and another list
(adjusted upon entry to and exit from each function) of local variables, for the collector’s
benefit.

The list of global variables is usually not too difficult to maintain, since global variables
are relatively rare. However, an explicitly maintained list of local variables (in the author’s
personal experience) is a nightmare to maintain. Thus, the BDW-GC uses a technique
called **conservative garbage collection**, to make the local variable list unnecessary.

The trick to conservative collection is to treat the stack as an ordinary range of memory,
and assume that *every* word on the stack is a pointer into the heap. Thus, the collector
marks all objects whose addresses appear anywhere in the stack, without knowing for sure
how that word is meant to be interpreted.

In addition to the stack, the BDW-GC will also scan static data sections. This means
that global variables are also scanned when looking for live Scheme objects.

Obviously, such a system will occasionally retain objects that are actually garbage, and
should be freed. In practice, this is not a problem. The alternative, an explicitly maintained
list of local variable addresses, is effectively much less reliable, due to programmer error.
Interested readers should see the BDW-GC web page at [http://www.hboehm.info/gc/](http://www.hboehm.info/gc/),
for more information.

### 9.2.5 The SCM Type in Guile

Guile classifies Scheme objects into two kinds: those that fit entirely within an SCM, and
those that require heap storage.

The former class are called **immediates**. The class of immediates includes small integers,
characters, boolean values, the empty list, the mysterious end-of-file object, and some others.

The remaining types are called, not surprisingly, **non-immediates**. They include pairs,
procedures, strings, vectors, and all other data types in Guile. For non-immediates, the SCM
word contains a pointer to data on the heap, with further information about the object in
question is stored in that data.

This section describes how the SCM type is actually represented and used at the C level.
Interested readers should see `libguile/tags.h` for an exposition of how Guile stores type
information.

In fact, there are two basic C data types to represent objects in Guile: SCM and `scm_t_bits`.

#### 9.2.5.1 Relationship between SCM and `scm_t_bits`

A variable of type SCM is guaranteed to hold a valid Scheme object. A variable of type
`scm_t_bits`, on the other hand, may hold a representation of a SCM value as a C integral
type, but may also hold any C value, even if it does not correspond to a valid Scheme object.

For a variable x of type SCM, the Scheme object’s type information is stored in a form
that is not directly usable. To be able to work on the type encoding of the scheme value,
the `SCM` variable has to be transformed into the corresponding representation as a `scm_t_bits` variable `y` by using the `SCM_UNPACK` macro. Once this has been done, the type of the scheme object `x` can be derived from the content of the bits of the `scm_t_bits` value `y`, in the way illustrated by the example earlier in this chapter (see Section 9.2.3 [Cheaper Pairs], page 806). Conversely, a valid bit encoding of a Scheme value as a `scm_t_bits` variable can be transformed into the corresponding `SCM` value using the `SCM_PACK` macro.

### 9.2.5.2 Immediate objects

A Scheme object may either be an immediate, i.e. carrying all necessary information by itself, or it may contain a reference to a cell with additional information on the heap. Although in general it should be irrelevant for user code whether an object is an immediate or not, within Guile’s own code the distinction is sometimes of importance. Thus, the following low level macro is provided:

```c
int SCM_IMP (SCM x) {
    A Scheme object is an immediate if it fulfills the SCM_IMP predicate, otherwise it holds an encoded reference to a heap cell. The result of the predicate is delivered as a C style boolean value. User code and code that extends Guile should normally not be required to use this macro.
}
```

Summary:
- Given a Scheme object `x` of unknown type, check first with `SCM_IMP (x)` if it is an immediate object.
- If so, all of the type and value information can be determined from the `scm_t_bits` value that is delivered by `SCM_UNPACK (x)`.

There are a number of special values in Scheme, most of them documented elsewhere in this manual. It’s not quite the right place to put them, but for now, here’s a list of the C names given to some of these values:

**SCM SCM_EOL**

The Scheme empty list object, or “End Of List” object, usually written in Scheme as `'()`.  

**SCM SCM_EOF_VAL**

The Scheme end-of-file value. It has no standard written representation, for obvious reasons.

**SCM SCM_UNSPECIFIED**

The value returned by some (but not all) expressions that the Scheme standard says return an “unspecified” value.

This is sort of a weirdly literal way to take things, but the standard read-eval-print loop prints nothing when the expression returns this value, so it’s not a bad idea to return this when you can’t think of anything else helpful.

**SCM SCM_UNDEFINED**

The “undefined” value. Its most important property is that is not equal to any valid Scheme value. This is put to various internal uses by C code interacting with Guile.
For example, when you write a C function that is callable from Scheme and which takes optional arguments, the interpreter passes `SCM_UNDEFINED` for any arguments you did not receive.

We also use this to mark unbound variables.

```c
int SCM_UNBNDP (SCM x)  
    [Macro]
    Return true if x is SCM_UNDEFINED. Note that this is not a check to see if x is SCM_UNBOUND. History will not be kind to us.
```

### 9.2.5.3 Non-immediate objects

A Scheme object of type `SCM` that does not fulfill the SCM_IMP predicate holds an encoded reference to a heap cell. This reference can be decoded to a C pointer to a heap cell using the `SCM2PTR` macro. The encoding of a pointer to a heap cell into a SCM value is done using the `PTR2SCM` macro.

```c
scm_t_cell * SCM2PTR (SCM x)  
    [Macro]
    Extract and return the heap cell pointer from a non-immediate SCM object x.

SCM PTR2SCM (scm_t_cell * x)  
    [Macro]
    Return a SCM value that encodes a reference to the heap cell pointer x.
```

Note that it is also possible to transform a non-immediate SCM value by using `SCM_UNPACK` into a `scm_t_bits` variable. However, the result of `SCM_UNPACK` may not be used as a pointer to a `scm_t_cell`: only `SCM2PTR` is guaranteed to transform a SCM object into a valid pointer to a heap cell. Also, it is not allowed to apply `PTR2SCM` to anything that is not a valid pointer to a heap cell.

Summary:

- Only use `SCM2PTR` on SCM values for which SCM_IMP is false!
- Don’t use `(scm_t_cell *) SCM_UNPACK (x)`! Use `SCM2PTR (x)` instead!
- Don’t use `PTR2SCM` for anything but a cell pointer!

### 9.2.5.4 Allocating Cells

Guile provides both ordinary cells with two slots, and double cells with four slots. The following two function are the most primitive way to allocate such cells.

If the caller intends to use it as a header for some other type, she must pass an appropriate magic value in `word.0`, to mark it as a member of that type, and pass whatever value as `word.1`, etc that the type expects. You should generally not need these functions, unless you are implementing a new datatype, and thoroughly understand the code in `<libguile/tags.h>`.

If you just want to allocate pairs, use `scm_cons`.

```c
SCM scm_cell (scm_t_bits word_0, scm_t_bits word_1)  
    [Function]
    Allocate a new cell, initialize the two slots with `word.0` and `word.1`, and return it.
    Note that `word.0` and `word.1` are of type `scm_t_bits`. If you want to pass a SCM object, you need to use SCM_UNPACK.
```
SCM scm_double_cell (scm_t_bits word_0, scm_t_bits word_1, scm_t_bits word_2, scm_t_bits word_3)  
Like scm_cell, but allocates a double cell with four slots.

9.2.5.5 Heap Cell Type Information

Heap cells contain a number of entries, each of which is either a scheme object of type SCM or a raw C value of type scm_t_bits. Which of the cell entries contain Scheme objects and which contain raw C values is determined by the first entry of the cell, which holds the cell type information.

scm_t_bits SCM_CELL_TYPE (SCM x)  
For a non-immediate Scheme object x, deliver the content of the first entry of the heap cell referenced by x. This value holds the information about the cell type.

void SCM_SET_CELL_TYPE (SCM x, scm_t_bits t)  
For a non-immediate Scheme object x, write the value t into the first entry of the heap cell referenced by x. The value t must hold a valid cell type.

9.2.5.6 Accessing Cell Entries

For a non-immediate Scheme object x, the object type can be determined by reading the cell type entry using the SCM_CELL_TYPE macro. For each different type of cell it is known which cell entries hold Scheme objects and which cell entries hold raw C data. To access the different cell entries appropriately, the following macros are provided.

scm_t_bits SCM_CELL_WORD (SCM x, unsigned int n)  
Deliver the cell entry n of the heap cell referenced by the non-immediate Scheme object x as raw data. It is illegal, to access cell entries that hold Scheme objects by using these macros. For convenience, the following macros are also provided.

- SCM_CELL_WORD_0 (x) ⇒ SCM_CELL_WORD (x, 0)
- SCM_CELL_WORD_1 (x) ⇒ SCM_CELL_WORD (x, 1)
- ...
- SCM_CELL_WORD_n (x) ⇒ SCM_CELL_WORD (x, n)

SCM SCM_CELL_OBJECT (SCM x, unsigned int n)  
Deliver the cell entry n of the heap cell referenced by the non-immediate Scheme object x as a Scheme object. It is illegal, to access cell entries that do not hold Scheme objects by using these macros. For convenience, the following macros are also provided.

- SCM_CELL_OBJECT_0 (x) ⇒ SCM_CELL_OBJECT (x, 0)
- SCM_CELL_OBJECT_1 (x) ⇒ SCM_CELL_OBJECT (x, 1)
- ...
- SCM_CELL_OBJECT_n (x) ⇒ SCM_CELL_OBJECT (x, n)

void SCM_SET_CELL_WORD (SCM x, unsigned int n, scm_t_bits w)  
Write the raw C value w into entry number n of the heap cell referenced by the non-immediate Scheme value x. Values that are written into cells this way may only
be read from the cells using the SCM_CELL_WORD macros or, in case cell entry 0 is written, using the SCM_CELL_TYPE macro. For the special case of cell entry 0 it has to be made sure that \( w \) contains a cell type information which does not describe a Scheme object. For convenience, the following macros are also provided.

- SCM_SET_CELL_WORD_0 \((x, w) \Rightarrow SCM\_SET\_CELL\_WORD (x, 0, w)\)
- SCM_SET_CELL_WORD_1 \((x, w) \Rightarrow SCM\_SET\_CELL\_WORD (x, 1, w)\)
- \( \ldots \)
- SCM_SET_CELL_WORD_n \((x, w) \Rightarrow SCM\_SET\_CELL\_WORD (x, n, w)\)

\[\text{void SCM_SET_CELL_OBJECT} (SCM x, unsigned int n, SCM o) \quad \text{[Macro]}\]
Write the Scheme object \( o \) into entry number \( n \) of the heap cell referenced by the non-immediate Scheme value \( x \). Values that are written into cells this way may only be read from the cells using the SCM_CELL_OBJECT macros or, in case cell entry 0 is written, using the SCM_CELL_TYPE macro. For the special case of cell entry 0 the writing of a Scheme object into this cell is only allowed if the cell forms a Scheme pair. For convenience, the following macros are also provided.

- SCM_SET_CELL_OBJECT_0 \((x, o) \Rightarrow SCM\_SET\_CELL\_OBJECT (x, 0, o)\)
- SCM_SET_CELL_OBJECT_1 \((x, o) \Rightarrow SCM\_SET\_CELL\_OBJECT (x, 1, o)\)
- \( \ldots \)
- SCM_SET_CELL_OBJECT_n \((x, o) \Rightarrow SCM\_SET\_CELL\_OBJECT (x, n, o)\)

Summary:
- For a non-immediate Scheme object \( x \) of unknown type, get the type information by using SCM_CELL_TYPE \((x)\).
- As soon as the cell type information is available, only use the appropriate access methods to read and write data to the different cell entries.

### 9.3 A Virtual Machine for Guile

Guile has both an interpreter and a compiler. To a user, the difference is transparent—interpreted and compiled procedures can call each other as they please.

The difference is that the compiler creates and interprets bytecode for a custom virtual machine, instead of interpreting the S-expressions directly. Loading and running compiled code is faster than loading and running source code.

The virtual machine that does the bytecode interpretation is a part of Guile itself. This section describes the nature of Guile’s virtual machine.

#### 9.3.1 Why a VM?

For a long time, Guile only had an interpreter. Guile’s interpreter operated directly on the S-expression representation of Scheme source code.

But while the interpreter was highly optimized and hand-tuned, it still performed many needless computations during the course of evaluating an expression. For example, application of a function to arguments needlessly consed up the arguments in a list. Evaluation of an expression always had to figure out what the car of the expression is—a procedure, a memoized form, or something else. All values have to be allocated on the heap. Et cetera.
The solution to this problem was to compile the higher-level language, Scheme, into a lower-level language for which all of the checks and dispatching have already been done—the code is instead stripped to the bare minimum needed to “do the job”.

The question becomes then, what low-level language to choose? There are many options. We could compile to native code directly, but that poses portability problems for Guile, as it is a highly cross-platform project.

So we want the performance gains that compilation provides, but we also want to maintain the portability benefits of a single code path. The obvious solution is to compile to a virtual machine that is present on all Guile installations.

The easiest (and most fun) way to depend on a virtual machine is to implement the virtual machine within Guile itself. Guile contains a bytecode interpreter (written in C) and a Scheme to bytecode compiler (written in Scheme). This way the virtual machine provides what Scheme needs (tail calls, multiple values, call/cc) and can provide optimized inline instructions for Guile (cons, struct-ref, etc.).

So this is what Guile does. The rest of this section describes that VM that Guile implements, and the compiled procedures that run on it.

Before moving on, though, we should note that though we spoke of the interpreter in the past tense, Guile still has an interpreter. The difference is that before, it was Guile’s main evaluator, and so was implemented in highly optimized C; now, it is actually implemented in Scheme, and compiled down to VM bytecode, just like any other program. (There is still a C interpreter around, used to bootstrap the compiler, but it is not normally used at runtime.)

The upside of implementing the interpreter in Scheme is that we preserve tail calls and multiple-value handling between interpreted and compiled code. The downside is that the interpreter in Guile 2.2 is still about twice as slow as the interpreter in 1.8. Since Scheme users are mostly running compiled code, the compiler’s speed more than makes up for the loss. In any case, once we have native compilation for Scheme code, we expect the self-hosted interpreter to handily beat the old hand-tuned C implementation.

Also note that this decision to implement a bytecode compiler does not preclude native compilation. We can compile from bytecode to native code at runtime, or even do ahead of time compilation. More possibilities are discussed in Section 9.4.7 [Extending the Compiler], page 855.

### 9.3.2 VM Concepts

Compiled code is run by a virtual machine (VM). Each thread has its own VM. The virtual machine executes the sequence of instructions in a procedure.

Each VM instruction starts by indicating which operation it is, and then follows by encoding its source and destination operands. Each procedure declares that it has some number of local variables, including the function arguments. These local variables form the available operands of the procedure, and are accessed by index.

The local variables for a procedure are stored on a stack. Calling a procedure typically enlarges the stack, and returning from a procedure shrinks it. Stack memory is exclusive to the virtual machine that owns it.
In addition to their stacks, virtual machines also have access to the global memory (modules, global bindings, etc) that is shared among other parts of Guile, including other VMs.

The registers that a VM has are as follows:

- **ip** - Instruction pointer
- **sp** - Stack pointer
- **fp** - Frame pointer

In other architectures, the instruction pointer is sometimes called the “program counter” (pc). This set of registers is pretty typical for virtual machines; their exact meanings in the context of Guile’s VM are described in the next section.

### 9.3.3 Stack Layout

The stack of Guile’s virtual machine is composed of frames. Each frame corresponds to the application of one compiled procedure, and contains storage space for arguments, local variables, and some bookkeeping information (such as what to do after the frame is finished).

While the compiler is free to do whatever it wants to, as long as the semantics of a computation are preserved, in practice every time you call a function, a new frame is created. (The notable exception of course is the tail call case, see Section 3.3.2 [Tail Calls], page 24.)

The structure of the top stack frame is as follows:

```
+--------------------------+ <- fp + 2 = SCM_FRAME_PREVIOUS_SP (fp)
| Dynamic link            |
+--------------------------+
| Return address           | <- fp
+--------------------------+
| Local 0                  |
+--------------------------+
| Local 1                  |
+--------------------------+
| ...                      |
+--------------------------+
| Local N-1                |
\----------------------------/ <- sp
```

In the above drawing, the stack grows downward. At the beginning of a function call, the procedure being applied is in local 0, followed by the arguments from local 1. After the procedure checks that it is being passed a compatible set of arguments, the procedure allocates some additional space in the frame to hold variables local to the function.

Note that once a value in a local variable slot is no longer needed, Guile is free to re-use that slot. This applies to the slots that were initially used for the callee and arguments, too. For this reason, backtraces in Guile aren’t always able to show all of the arguments: it could be that the slot corresponding to that argument was re-used by some other variable.

The *return address* is the **ip** that was in effect before this program was applied. When we return from this activation frame, we will jump back to this **ip**. Likewise, the *dynamic*
link is the offset of the fp that was in effect before this program was applied, relative to the current fp.

To prepare for a non-tail application, Guile’s VM will emit code that shuffles the function to apply and its arguments into appropriate stack slots, with two free slots below them. The call then initializes those free slots with the current ip and fp, and updates ip to point to the function entry, and fp to point to the new call frame.

In this way, the dynamic link links the current frame to the previous frame. Computing a stack trace involves traversing these frames.

Each stack local in Guile is 64 bits wide, even on 32-bit architectures. This allows Guile to preserve its uniform treatment of stack locals while allowing for unboxed arithmetic on 64-bit integers and floating-point numbers. See Section 9.3.7 [Instruction Set], page 819, for more on unboxed arithmetic.

As an implementation detail, we actually store the dynamic link as an offset and not an absolute value because the stack can move at runtime as it expands or during partial continuation calls. If it were an absolute value, we would have to walk the frames, relocating frame pointers.

9.3.4 Variables and the VM

Consider the following Scheme code as an example:

```scheme
(define (foo a)
  (lambda (b) (list foo a b)))
```

Within the lambda expression, foo is a top-level variable, a is a lexically captured variable, and b is a local variable.

Another way to refer to a and b is to say that a is a “free” variable, since it is not defined within the lambda, and b is a “bound” variable. These are the terms used in the lambda calculus, a mathematical notation for describing functions. The lambda calculus is useful because it is a language in which to reason precisely about functions and variables. It is especially good at describing scope relations, and it is for that reason that we mention it here.

Guile allocates all variables on the stack. When a lexically enclosed procedure with free variables—a closure—is created, it copies those variables into its free variable vector. References to free variables are then redirected through the free variable vector.

If a variable is ever set!, however, it will need to be heap-allocated instead of stack-allocated, so that different closures that capture the same variable can see the same value. Also, this allows continuations to capture a reference to the variable, instead of to its value at one point in time. For these reasons, set! variables are allocated in “boxes”—actually, in variable cells. See Section 6.20.7 [Variables], page 415, for more information. References to set! variables are indirected through the boxes.

Thus perhaps counterintuitively, what would seem “closer to the metal”, viz set!, actually forces an extra memory allocation and indirection. Sometimes Guile’s optimizer can remove this allocation, but not always.

Going back to our example, b may be allocated on the stack, as it is never mutated. a may also be allocated on the stack, as it too is never mutated. Within the enclosed lambda, its value will be copied into (and referenced from) the free variables vector.

foo is a top-level variable, because foo is not lexically bound in this example.
9.3.5 Compiled Procedures are VM Programs

By default, when you enter in expressions at Guile’s REPL, they are first compiled to bytecode. Then that bytecode is executed to produce a value. If the expression evaluates to a procedure, the result of this process is a compiled procedure.

A compiled procedure is a compound object consisting of its bytecode and a reference to any captured lexical variables. In addition, when a procedure is compiled, it has associated metadata written to side tables, for instance a line number mapping, or its docstring. You can pick apart these pieces with the accessors in (system vm program). See Section 6.9.3 [Compiled Procedures], page 250, for a full API reference.

A procedure may reference data that was statically allocated when the procedure was compiled. For example, a pair of immediate objects (see Section 9.2.5.2 [Immediate objects], page 809) can be allocated directly in the memory segment that contains the compiled bytecode, and accessed directly by the bytecode.

Another use for statically allocated data is to serve as a cache for a bytecode. Top-level variable lookups are handled in this way. If the toplevel-box instruction finds that it does not have a cached variable for a top-level reference, it accesses other static data to resolve the reference, and fills in the cache slot. Thereafter all access to the variable goes through the cache cell. The variable’s value may change in the future, but the variable itself will not.

We can see how these concepts tie together by disassembling the foo function we defined earlier to see what is going on:

```
scheme@(guile-user)> (define (foo a) (lambda (b) (list foo a b)))
scheme@(guile-user)> ,x foo
```

Disassembly of #<procedure foo (a)> at #xea4ce4:

```
0  (assert-nargs-ee/locals 2 0) ;; 2 slots (1 arg) at (unknown file):1:0
1  (make-closure 1 7 1) ;; anonymous procedure at #xea4d04 (1 free var)
4  (free-set! 1 0 0) ;; free var 0
6  (mov 0 1)
7  (return-values 2) ;; 1 value
```

Disassembly of anonymous procedure at #xea4d04:

```
0  (assert-nargs-ee/locals 2 2) ;; 4 slots (1 arg) at (unknown file):1:16
1  (toplevel-box 1 74 58 68 #t) ;; ‘foo’
6  (box-ref 1 1)
7  (make-short-immediate 0 772) ;; () at (unknown file):1:28
8  (cons 2 2 0)
9  (free-ref 3 3 0) ;; free var 0
11 (cons 3 3 2)
12 (cons 2 1 3)
13 (return-values 2) ;; 1 value
```

First there’s some prelude, where foo checks that it was called with only 1 argument. Then at ip 1, we allocate a new closure and store it in slot 1, relative to the sp.

At run-time, local variables in Guile are usually addressed relative to the stack pointer, which leads to a pleasantly efficient sp[n] access. However it can make the disassembly hard to read, because the sp can change during the function, and because incoming arguments are relative to the fp, not the sp.
To know what fp-relative slot corresponds to an sp-relative reference, scan up in the disassembly until you get to a “n slots” annotation; in our case, 2, indicating that the frame has space for 2 slots. Thus a zero-indexed sp-relative slot of 1 corresponds to the fp-relative slot of 0, which initially held the value of the closure being called. This means that Guile doesn’t need the value of the closure to compute its result, and so slot 0 was free for re-use, in this case for the result of making a new closure.

A closure is code with data. The 6 in the `(make-closure 1 6 1)` is a relative offset from the instruction pointer of the code for the closure, and the final 1 indicates that the closure has space for 1 free variable. Ip 4 initializes free variable 0 in the new closure with the value from sp-relative slot 0, which corresponds to fp-relative slot 1, the first argument of foo: a. Finally we return the closure.

The second stanza disassembles the code for the closure. After the prelude, we load the variable for the toplevel variable foo into slot 1. This lookup occurs lazily, the first time the variable is actually referenced, and the location of the lookup is cached so that future references are very cheap. See Section 9.3.7.2 [Top-Level Environment Instructions], page 821, for more details. The box-ref dereferences the variable cell, replacing the contents of slot 1.

What follows is a sequence of conses to build up the result list. Ip 7 makes the tail of the list. Ip 8 conses on the value in slot 2, corresponding to the first argument to the closure: b. Ip 9 loads free variable 0 of slot 3 – the procedure being called, in fp-relative slot 0 – into slot 3, then ip 11 conses it onto the list. Finally we cons the value in slot 1, containing the foo toplevel, onto the front of the list, and we return it.

### 9.3.6 Object File Format

To compile a file to disk, we need a format in which to write the compiled code to disk, and later load it into Guile. A good object file format has a number of characteristics:

- Above all else, it should be very cheap to load a compiled file.
- It should be possible to statically allocate constants in the file. For example, a bytevector literal in source code can be emitted directly into the object file.
- The compiled file should enable maximum code and data sharing between different processes.
- The compiled file should contain debugging information, such as line numbers, but that information should be separated from the code itself. It should be possible to strip debugging information if space is tight.

These characteristics are not specific to Scheme. Indeed, mainstream languages like C and C++ have solved this issue many times in the past. Guile builds on their work by adopting ELF, the object file format of GNU and other Unix-like systems, as its object file format. Although Guile uses ELF on all platforms, we do not use platform support for ELF. Guile implements its own linker and loader. The advantage of using ELF is not sharing code, but sharing ideas. ELF is simply a well-designed object file format.

An ELF file has two meta-tables describing its contents. The first meta-table is for the loader, and is called the program table or sometimes the segment table. The program table divides the file into big chunks that should be treated differently by the loader. Mostly the difference between these segments is their permissions.
Typically all segments of an ELF file are marked as read-only, except that part that represents modifiable static data or static data that needs load-time initialization. Loading an ELF file is as simple as mmapping the thing into memory with read-only permissions, then using the segment table to mark a small sub-region of the file as writable. This writable section is typically added to the root set of the garbage collector as well.

One ELF segment is marked as “dynamic”, meaning that it has data of interest to the loader. Guile uses this segment to record the Guile version corresponding to this file. There is also an entry in the dynamic segment that points to the address of an initialization thunk that is run to perform any needed link-time initialization. (This is like dynamic relocations for normal ELF shared objects, except that we compile the relocations as a procedure instead of having the loader interpret a table of relocations.) Finally, the dynamic segment marks the location of the “entry thunk” of the object file. This thunk is returned to the caller of `load-thunk-from-memory` or `load-thunk-from-file`. When called, it will execute the “body” of the compiled expression.

The other meta-table in an ELF file is the `section table`. Whereas the program table divides an ELF file into big chunks for the loader, the section table specifies small sections for use by introspective tools like debuggers or the like. One segment (program table entry) typically contains many sections. There may be sections outside of any segment, as well.

Typical sections in a Guile .go file include:

- `.rtl-text`
  Bytecode.

- `.data`
  Data that needs initialization, or which may be modified at runtime.

- `.rodata`
  Statically allocated data that needs no run-time initialization, and which therefore can be shared between processes.

- `.dynamic`
  The dynamic section, discussed above.

- `.symtab`
- `.strtab`
  A table mapping addresses in the `.rtl-text` to procedure names. `.strtab` is used by `.symtab`.

- `.guile.procprops`
- `.guile.arities`
- `.guile.arities.strtab`
- `.guile.docstrs`
- `.guile.docstrs.strtab`
  Side tables of procedure properties, arities, and docstrings.

- `.guile.docstrs.strtab`
  Side table of frame maps, describing the set of live slots for ever return point in the program text, and whether those slots are pointers are not. Used by the garbage collector.
9.3.7 Instruction Set

There are currently about 175 instructions in Guile’s virtual machine. These instructions represent atomic units of a program’s execution. Ideally, they perform one task without conditional branches, then dispatch to the next instruction in the stream.

Instructions themselves are composed of 1 or more 32-bit units. The low 8 bits of the first word indicate the opcode, and the rest of instruction describe the operands. There are a number of different ways operands can be encoded.

- \texttt{s}n: An unsigned \(n\)-bit integer, indicating the sp-relative index of a local variable.
- \texttt{f}n: An unsigned \(n\)-bit integer, indicating the fp-relative index of a local variable. Used when a continuation accepts a variable number of values, to shuffle received values into known locations in the frame.
- \texttt{c}n: An unsigned \(n\)-bit integer, indicating a constant value.
- \texttt{l}24: An offset from the current \texttt{ip}, in 32-bit units, as a signed 24-bit value. Indicates a bytecode address, for a relative jump.
- \texttt{i}16, \texttt{i}32: An immediate Scheme value (see Section 9.2.5.2 [Immediate objects], page 809), encoded directly in 16 or 32 bits.
- \texttt{a}32, \texttt{b}32: An immediate Scheme value, encoded as a pair of 32-bit words. \texttt{a}32 and \texttt{b}32 values always go together on the same opcode, and indicate the high and low bits, respectively. Normally only used on 64-bit systems.
- \texttt{n}32: A statically allocated non-immediate. The address of the non-immediate is encoded as a signed 32-bit integer, and indicates a relative offset in 32-bit units. Think of it as \(\texttt{SCM x = ip + offset}\).
- \texttt{r}32: Indirect scheme value, like \texttt{n}32 but indirected. Think of it as \(\texttt{SCM \star x = ip + offset}\).
lo32  An ip-relative address, as a signed 32-bit integer. Could indicate a bytecode address, as in make-closure, or a non-immediate address, as with static-patch!.

132 and lo32 are the same from the perspective of the virtual machine. The difference is that an assembler might want to allow an lo32 address to be specified as a label and then some number of words offset from that label, for example when patching a field of a statically allocated object.

b1  A boolean value: 1 for true, otherwise 0.

ox  An ignored sequence of \( n \) bits.

An instruction is specified by giving its name, then describing its operands. The operands are packed by 32-bit words, with earlier operands occupying the lower bits.

For example, consider the following instruction specification:

\texttt{free-set! s12:dst s12:src x8:_ c24:idx}  
\comment{Instruction}

Set free variable \( \text{id}x \) from the closure \( dst \) to \( src \).

The first word in the instruction will start with the 8-bit value corresponding to the \texttt{free-set!} opcode in the low bits, followed by \( dst \) and \( src \) as 12-bit values. The second word starts with 8 dead bits, followed by the index as a 24-bit immediate value.

Sometimes the compiler can figure out that it is compiling a special case that can be run more efficiently. So, for example, while Guile offers a generic test-and-branch instruction, it also offers specific instructions for special cases, so that the following cases all have their own test-and-branch instructions:

\begin{verbatim}
(if pred then else)
(if (not pred) then else)
(if (null? 1) then else)
(if (not (null? 1)) then else)
\end{verbatim}

In addition, some Scheme primitives have their own inline implementations. For example, in the previous section we saw \texttt{cons}.

Finally, for instructions with operands that encode references to the stack, the interpretation of those stack values is up to the instruction itself. Most instructions expect their operands to be tagged SCM values (\texttt{scm} representation), but some instructions expect unboxed integers (\texttt{u64} and \texttt{s64} representations) or floating-point numbers (\texttt{f64} representation). Instructions have static types: they must receive their operands in the format they expect. It’s up to the compiler to ensure this is the case. Unless otherwise mentioned, all operands and results are boxed as SCM values.

\subsection*{9.3.7.1 Lexical Environment Instructions}

These instructions access and mutate the lexical environment of a compiled procedure—its free and bound variables. See Section 9.3.3 [Stack Layout], page 814, for more information on the format of stack frames.

\texttt{mov s12:dst s12:src}  
\comment{Instruction}

\texttt{long-mov s24:dst x8:_ s24:src}  
\comment{Instruction}

Copy a value from one local slot to another.
As discussed previously, procedure arguments and local variables are allocated to local slots. Guile’s compiler tries to avoid shuffling variables around to different slots, which often makes \texttt{mov} instructions redundant. However there are some cases in which shuffling is necessary, and in those cases, \texttt{mov} is the thing to use.

\texttt{long-fmov \_f24:dst x8:\_f24:src} \hfill \textbf{[Instruction]}

Copy a value from one local slot to another, but addressing slots relative to the \texttt{fp} instead of the \texttt{sp}. This is used when shuffling values into place after multiple-value returns.

\texttt{make-closure s24:dst l32:offset x8:\_c24:nfree} \hfill \textbf{[Instruction]}

Make a new closure, and write it to \texttt{dst}. The code for the closure will be found at \texttt{offset} words from the current \texttt{ip}. \texttt{offset} is a signed 32-bit integer. Space for \texttt{nfree} free variables will be allocated.

The size of a closure is currently two words, plus one word per free variable.

\texttt{free-ref s12:dst s12:src x8:\_c24:idx} \hfill \textbf{[Instruction]}

Load free variable \texttt{idx} from the closure \texttt{src} into local slot \texttt{dst}.

\texttt{free-set! s12:dst s12:src x8:\_c24:idx} \hfill \textbf{[Instruction]}

Set free variable \texttt{idx} from the closure \texttt{dst} to \texttt{src}.

This instruction is usually used when initializing a closure’s free variables, but not to mutate free variables, as variables that are assigned are boxed.

Recall that variables that are assigned are usually allocated in boxes, so that continuations and closures can capture their identity and not their value at one point in time. Variables are also used in the implementation of top-level bindings; see the next section for more information.

\texttt{box s12:dst s12:src} \hfill \textbf{[Instruction]}

Create a new variable holding \texttt{src}, and place it in \texttt{dst}.

\texttt{box-ref s12:dst s12:src} \hfill \textbf{[Instruction]}

Unpack the variable at \texttt{src} into \texttt{dst}, asserting that the variable is actually bound.

\texttt{box-set! s12:dst s12:src} \hfill \textbf{[Instruction]}

Set the contents of the variable at \texttt{dst} to \texttt{set}.

\subsection*{9.3.7.2 Top-Level Environment Instructions}

These instructions access values in the top-level environment: bindings that were not lexically apparent at the time that the code in question was compiled.

The location in which a toplevel binding is stored can be looked up once and cached for later. The binding itself may change over time, but its location will stay constant.

\texttt{current-module s24:dst} \hfill \textbf{[Instruction]}

Store the current module in \texttt{dst}.

\texttt{resolve s24:dst b1:bound? x7:\_s24:sym} \hfill \textbf{[Instruction]}

Resolve \texttt{sym} in the current module, and place the resulting variable in \texttt{dst}. An error will be signalled if no variable is found. If \texttt{bound?} is true, an error will be signalled if the variable is unbound.
**define! s12:dst s12:sym**  
Look up a binding for sym in the current module, creating it if necessary. Store that variable to dst.

**toplevel-box s24:dst r32:var-offset r32:mod-offset**  
*n32:sym-offset b1:bound? x31:_*  
Load a value. The value will be fetched from memory, var-offset 32-bit words away from the current instruction pointer. var-offset is a signed value. Up to here, toplevel-box is like static-ref.

Then, if the loaded value is a variable, it is placed in dst, and control flow continues. Otherwise, we have to resolve the variable. In that case we load the module from mod-offset, just as we loaded the variable. Usually the module gets set when the closure is created. sym-offset specifies the name, as an offset to a symbol.

We use the module and the symbol to resolve the variable, placing it in dst, and caching the resolved variable so that we will hit the cache next time. If bound? is true, an error will be signalled if the variable is unbound.

**module-box s24:dst r32:var-offset n32:mod-offset**  
*n32:sym-offset b1:bound? x31:_*  
Like toplevel-box, except mod-offset points at a module identifier instead of the module itself. A module identifier is a module name, as a list, prefixed by a boolean. If the prefix is true, then the variable is resolved relative to the module’s public interface instead of its private interface.

### 9.3.7.3 Procedure Call and Return Instructions

As described earlier (see Section 9.3.3 [Stack Layout], page 814), Guile’s calling convention is that arguments are passed and values returned on the stack.

For calls, both in tail position and in non-tail position, we require that the procedure and the arguments already be shuffled into place before the call instruction. “Into place” for a tail call means that the procedure should be in slot 0, relative to the fp, and the arguments should follow. For a non-tail call, if the procedure is in fp-relative slot n, the arguments should follow from slot n+1, and there should be two free slots at n-1 and n-2 in which to save the ip and fp.

Returning values is similar. Multiple-value returns should have values already shuffled down to start from fp-relative slot 1 before emitting return-values. We start from slot 1 instead of slot 0 to make tail calls to values trivial.

In both calls and returns, the sp is used to indicate to the callee or caller the number of arguments or return values, respectively. After receiving return values, it is the caller’s responsibility to restore the frame by resetting the sp to its former value.

**call f24:proc x8:_ c24:nlocals**  
Call a procedure. proc is the local corresponding to a procedure. The two values below proc will be overwritten by the saved call frame data. The new frame will have space for nlocals locals: one for the procedure, and the rest for the arguments which should already have been pushed on.
When the call returns, execution proceeds with the next instruction. There may be any number of values on the return stack; the precise number can be had by subtracting the address of proc from the post-call sp.

**call-label f24:proc x8:_ c24:nlocals l32:label**  
Call a procedure in the same compilation unit.

This instruction is just like call, except that instead of dereferencing proc to find the call target, the call target is known to be at label, a signed 32-bit offset in 32-bit units from the current ip. Since proc is not dereferenced, it may be some other representation of the closure.

**tail-call c24:nlocals**  
Tail-call a procedure. Requires that the procedure and all of the arguments have already been shuffled into position. Will reset the frame to nlocals.

**tail-call-label c24:nlocals l32:label**  
Tail-call a known procedure. As call is to call-label, tail-call is to tail-call-label.

**tail-call/shuffle f24:from**  
Tail-call a procedure. The procedure should already be set to slot 0. The rest of the args are taken from the frame, starting at from, shuffled down to start at slot 0. This is part of the implementation of the call-with-values builtin.

**receive f12:dst f12:proc x8:_ c24:nlocals**  
Receive a single return value from a call whose procedure was in proc, asserting that the call actually returned at least one value. Afterwards, resets the frame to nlocals locals.

**receive-values f24:proc b1:allow-extra? x7:_ c24:nvalues**  
Receive a return of multiple values from a call whose procedure was in proc. If fewer than nvalues values were returned, signal an error. Unless allow-extra? is true, require that the number of return values equals nvalues exactly. After receive-values has run, the values can be copied down via mov, or used in place.

**return-values c24:nlocals**  
Return a number of values from a call frame. This opcode corresponds to an application of values in tail position. As with tail calls, we expect that the values have already been shuffled down to a contiguous array starting at slot 1. If nlocals is nonzero, reset the frame to hold that number of locals. Note that a frame reset to 1 local returns 0 values.

**call/cc x24:_**  
Capture the current continuation, and tail-apply the procedure in local slot 1 to it. This instruction is part of the implementation of call/cc, and is not generated by the compiler.
9.3.7.4 Function Prologue Instructions

A function call in Guile is very cheap: the VM simply hands control to the procedure. The procedure itself is responsible for asserting that it has been passed an appropriate number of arguments. This strategy allows arbitrarily complex argument parsing idioms to be developed, without harming the common case.

For example, only calls to keyword-argument procedures “pay” for the cost of parsing keyword arguments. (At the time of this writing, calling procedures with keyword arguments is typically two to four times as costly as calling procedures with a fixed set of arguments.)

(assert-nargs-ee \texttt{c24:expected})

(assert-nargs-ge \texttt{c24:expected})

(assert-nargs-le \texttt{c24:expected})

If the number of actual arguments is not \texttt{==}, \texttt{>=}, or \texttt{<=} \texttt{expected}, respectively, signal an error.

The number of arguments is determined by subtracting the stack pointer from the frame pointer (\texttt{fp - sp}). See Section 9.3.3 [Stack Layout], page 814, for more details on stack frames. Note that \texttt{expected} includes the procedure itself.

(br-if-nargs-ne \texttt{c24:expected} \texttt{x8:_ l24:offset})

(br-if-nargs-lt \texttt{c24:expected} \texttt{x8:_ l24:offset})

(br-if-nargs-gt \texttt{c24:expected} \texttt{x8:_ l24:offset})

If the number of actual arguments is not equal, less than, or greater than \texttt{expected}, respectively, add \texttt{offset}, a signed 24-bit number, to the current instruction pointer. Note that \texttt{expected} includes the procedure itself.

These instructions are used to implement multiple arities, as in \texttt{case-lambda}. See Section 6.9.5 [Case-lambda], page 255, for more information.

(alloc-frame \texttt{c24:nlocals})

Ensure that there is space on the stack for \texttt{nlocals} local variables, setting them all to \texttt{SCM_UNDEFINED}, except those values that are already on the stack.

(reset-frame \texttt{c24:nlocals})

Like \texttt{alloc-frame}, but doesn’t check that the stack is big enough, and doesn’t initialize values to \texttt{SCM_UNDEFINED}. Used to reset the frame size to something less than the size that was previously set via alloc-frame.

(assert-nargs-ee/locals \texttt{c12:expected} \texttt{c12:nlocals})

Equivalent to a sequence of \texttt{assert-nargs-ee} and \texttt{reserve-locals}. The number of locals reserved is \texttt{expected} + \texttt{nlocals}.

(br-if-npos-gt \texttt{c24:nreq} \texttt{x8:_ c24:npos} \texttt{x8:_ l24:offset})

Find the first positional argument after \texttt{nreq}. If it is greater than \texttt{npos}, jump to \texttt{offset}.

This instruction is only emitted for functions with multiple clauses, and an earlier clause has keywords and no rest arguments. See Section 6.9.5 [Case-lambda], page 255, for more on how \texttt{case-lambda} chooses the clause to apply.
bind-kwargs c24:nreq c8:flags c24:nreq-and-opt x8:_

flags is a bitfield, whose lowest bit is allow-other-keys, second bit is has-rest, and whose following six bits are unused.

Find the last positional argument, and shuffle all the rest above ntotal. Initialize the intervening locals to SCM_UNDEFINED. Then load the constant at kw-offset words from the current ip, and use it and the allow-other-keys flag to bind keyword arguments. If has-rest, collect all shuffled arguments into a list, and store it in nreq-and-opt. Finally, clear the arguments that we shuffled up.

The parsing is driven by a keyword arguments association list, looked up using kw-offset. The alist is a list of pairs of the form (kw . index), mapping keyword arguments to their local slot indices. Unless allow-other-keys is set, the parser will signal an error if an unknown key is found.

A macro-mega-instruction.

bind-rest f24:dst

Collect any arguments at or above dst into a list, and store that list at dst.

9.3.7.5 Trampoline Instructions

Though most applicable objects in Guile are procedures implemented in bytecode, not all are. There are primitives, continuations, and other procedure-like objects that have their own calling convention. Instead of adding special cases to the call instruction, Guile wraps these other applicable objects in VM trampoline procedures, then provides special support for these objects in bytecode.

Trampoline procedures are typically generated by Guile at runtime, for example in response to a call to scm_c_make_gsubr. As such, a compiler probably shouldn’t emit code with these instructions. However, it’s still interesting to know how these things work, so we document these trampoline instructions here.

subr-call x24:_

Call a subr, passing all locals in this frame as arguments. Return from the calling frame.

foreign-call c12:cif-idx c12:ptr-idx

Call a foreign function. Fetch the cif and foreign pointer from cif-idx and ptr-idx, both free variables. Return from the calling frame. Arguments are taken from the stack.

continuation-call c24:contregs

Return to a continuation, nonlocally. The arguments to the continuation are taken from the stack. contregs is a free variable containing the reified continuation.

compose-continuation c24:cont

Compose a partial continuation with the current continuation. The arguments to the continuation are taken from the stack. cont is a free variable containing the reified continuation.
tail-apply x24:_
Tail-apply the procedure in local slot 0 to the rest of the arguments. This instruction is part of the implementation of apply, and is not generated by the compiler.

builtin-ref s12:dst c12:idx
Load a builtin stub by index into dst.

apply-non-program x24:_
An instruction used only by a special trampoline that the VM uses to apply non-programs. Using that trampoline allows profilers and backtrace utilities to avoid seeing the instruction pointer from the calling frame.

9.3.7.6 Branch Instructions
All offsets to branch instructions are 24-bit signed numbers, which count 32-bit units. This gives Guile effectively a 26-bit address range for relative jumps.

br l24:offset
Add offset to the current instruction pointer.

All the conditional branch instructions described below have an invert parameter, which if true reverses the test: br-if-true becomes br-if-false, and so on.

br-if-true s24:test b1:invert x7:_ l24:offset
If the value in test is true for the purposes of Scheme, add offset to the current instruction pointer.

br-if-null s24:test b1:invert x7:_ l24:offset
If the value in test is the end-of-list or Lisp nil, add offset to the current instruction pointer.

br-if-nil s24:test b1:invert x7:_ l24:offset
If the value in test is false to Lisp, add offset to the current instruction pointer.

br-if-pair s24:test b1:invert x7:_ l24:offset
If the value in test is a pair, add offset to the current instruction pointer.

br-if-struct s24:test b1:invert x7:_ l24:offset
If the value in test is a struct, add offset number to the current instruction pointer.

br-if-char s24:test b1:invert x7:_ l24:offset
If the value in test is a char, add offset to the current instruction pointer.

br-if-tc7 s24:test b1:invert u7:tc7 l24:offset
If the value in test has the TC7 given in the second word, add offset to the current instruction pointer. TC7 codes are part of the way Guile represents non-immediate objects, and are deep wizardry. See libguile/tags.h for all the details.

br-if-eq s24:a x8:_ s24:b b1:invert x7:_ l24:offset
br-if-eqv s24:a x8:_ s24:b b1:invert x7:_ l24:offset
If the value in a is eq? or eqv? to the value in b, respectively, add offset to the current instruction pointer.
Chapter 9: Guile Implementation

9.3.7.7 Constant Instructions

The following instructions load literal data into a program. There are two kinds.

The first set of instructions loads immediate values. These instructions encode the immediate directly into the instruction stream.

- **make-short-immediate** `s8:dst i16:low-bits`  
  Make an immediate whose low bits are `low-bits`, and whose top bits are 0.

- **make-long-immediate** `s24:dst i32:low-bits`  
  Make an immediate whose low bits are `low-bits`, and whose top bits are 0.

- **make-long-long-immediate** `s24:dst a32:high-bits b32:low-bits`  
  Make an immediate with `high-bits` and `low-bits`.

Non-immediate constant literals are referenced either directly or indirectly. For example, Guile knows at compile-time what the layout of a string will be like, and arranges to embed that object directly in the compiled image. A reference to a string will use **make-non-immediate** to treat a pointer into the compilation unit as a SCM value directly.

- **make-non-immediate** `s24:dst n32:offset`  
  Load a pointer to statically allocated memory into `dst`. The object’s memory is found 32-bit words away from the current instruction pointer. Whether the object is mutable or immutable depends on where it was allocated by the compiler, and loaded by the loader.

Some objects must be unique across the whole system. This is the case for symbols and keywords. For these objects, Guile arranges to initialize them when the compilation unit is loaded, storing them into a slot in the image. References go indirectly through that slot. **static-ref** is used in this case.

- **static-ref** `s24:dst r32:offset`  
  Load a scm value into `dst`. The scm value will be fetched from memory, `offset` 32-bit words away from the current instruction pointer. `offset` is a signed value.

Fields of non-immediates may need to be fixed up at load time, because we do not know in advance at what address they will be loaded. This is the case, for example, for a pair containing a non-immediate in one of its fields. **static-ref** and **static-patch!** are used in these situations.
**static-set! s24:src lo32:offset**

Store a scm value into memory, offset 32-bit words away from the current instruction pointer. offset is a signed value.

**static-patch! x24: _ lo32:dst-offset l32:src-offset**

Patch a pointer at dst-offset to point to src-offset. Both offsets are signed 32-bit values, indicating a memory address as a number of 32-bit words away from the current instruction pointer.

Many kinds of literals can be loaded with the above instructions, once the compiler has prepared the statically allocated data. This is the case for vectors, strings, uniform vectors, pairs, and procedures with no free variables. Other kinds of data might need special initializers; those instructions follow.

**string->number s12:dst s12:src**

Parse a string in src to a number, and store in dst.

**string->symbol s12:dst s12:src**

Parse a string in src to a symbol, and store in dst.

**symbol->keyword s12:dst s12:src**

Make a keyword from the symbol in src, and store it in dst.

**load-typed-array s24:dst x8:_ s24:type x8:_ s24:shape n32:offset u32:len**

Load the contiguous typed array located at offset 32-bit words away from the instruction pointer, and store into dst. len is a byte length. offset is signed.

### 9.3.7.8 Dynamic Environment Instructions

Guile’s virtual machine has low-level support for **dynamic-wind**, dynamic binding, and composable prompts and aborts.

**abort x24:_**

Abort to a prompt handler. The tag is expected in slot 1, and the rest of the values in the frame are returned to the prompt handler. This corresponds to a tail application of abort-to-prompt.

If no prompt can be found in the dynamic environment with the given tag, an error is signalled. Otherwise all arguments are passed to the prompt’s handler, along with the captured continuation, if necessary.

If the prompt’s handler can be proven to not reference the captured continuation, no continuation is allocated. This decision happens dynamically, at run-time; the general case is that the continuation may be captured, and thus resumed. A reinstated continuation will have its arguments pushed on the stack from slot 1, as if from a multiple-value return, and control resumes in the caller. Thus to the calling function, a call to abort-to-prompt looks like any other function call.

**prompt s24:tag b1:escape-only? x7:_ f24:proc-slot x8:_ l24:handler-offset**

Push a new prompt on the dynamic stack, with a tag from tag and a handler at handler-offset words from the current ip.
If an abort is made to this prompt, control will jump to the handler. The handler will expect a multiple-value return as if from a call with the procedure at proc-slot, with the reified partial continuation as the first argument, followed by the values returned to the handler. If control returns to the handler, the prompt is already popped off by the abort mechanism. (Guile’s prompt implements Felleisen’s –F– operator.)

If escape-only? is nonzero, the prompt will be marked as escape-only, which allows an abort to this prompt to avoid reifying the continuation.

See Section 6.13.5 [Prompts], page 303, for more information on prompts.

wind s12:winder s12:unwinder

Push wind and unwind procedures onto the dynamic stack. Note that neither are actually called; the compiler should emit calls to wind and unwind for the normal dynamic-wind control flow. Also note that the compiler should have inserted checks that they wind and unwind procs are thunks, if it could not prove that to be the case. See Section 6.13.10 [Dynamic Wind], page 317.

unwind x24:_

a normal exit from the dynamic extent of an expression. Pop the top entry off of the dynamic stack.

push-fluid s12:fluid s12:value

Dynamically bind value to fluid by creating a with-fluids object and pushing that object on the dynamic stack. See Section 6.13.11 [Fluids and Dynamic States], page 321.

pop-fluid x24:_

Leave the dynamic extent of a with-fluid* expression, restoring the fluid to its previous value. push-fluid should always be balanced with pop-fluid.

fluid-ref s12:dst s12:src

Reference the fluid in src, and place the value in dst.

fluid-set! s12:fluid s12:val

Set the value of the fluid in dst to the value in src.

current-thread s24:dst

Write the value of the current thread to dst.

push-dynamic-state s24:state

Save the current set of fluid bindings on the dynamic stack and instate the bindings from state instead. See Section 6.13.11 [Fluids and Dynamic States], page 321.

pop-dynamic-state x24:_

Restore a saved set of fluid bindings from the dynamic stack. push-dynamic-state should always be balanced with pop-dynamic-state.

9.3.7.9 Miscellaneous Instructions

halt x24:_

Bring the VM to a halt, returning all the values from the stack. Used in the “boot continuation”, which is used when entering the VM from C.
push s24:src  
Bump the stack pointer by one word, and fill it with the value from slot src. The offset to src is calculated before the stack pointer is adjusted.

The push instruction is used when another instruction is unable to address an operand because the operand is encoded with fewer than 24 bits. In that case, Guile’s assembler will transparently emit code that temporarily pushes any needed operands onto the stack, emits the original instruction to address those now-near variables, then shuffles the result (if any) back into place.

pop s24:dst  
Pop the stack pointer, storing the value that was there in slot dst. The offset to dst is calculated after the stack pointer is adjusted.

drop c24:count  
Pop the stack pointer by count words, discarding any values that were stored there.

handle-interrupts x24:_  
Handle pending asynchronous interrupts (asyncs). See Section 6.22.3 [Asyns], page 439. The compiler inserts handle-interrupts instructions before any call, return, or loop back-edge.

return-from-interrupt x24:_  
A special instruction to return from a call and also pop off the stack frame from the call. Used when returning from asynchronous interrupts.

9.3.7.10 Inlined Scheme Instructions

The Scheme compiler can recognize the application of standard Scheme procedures. It tries to inline these small operations to avoid the overhead of creating new stack frames. This allows the compiler to optimize better.

make-vector s8:dst s8:length s8:init  
Make a vector and write it to dst. The vector will have space for length slots. They will be filled with the value in slot init.

make-vector/immediate s8:dst s8:length c8:init  
Make a short vector of known size and write it to dst. The vector will have space for length slots, an immediate value. They will be filled with the value in slot init.

vector-length s12:dst s12:src  
Store the length of the vector in src in dst, as an unboxed unsigned 64-bit integer.

vector-ref s8:dst s8:src s8:idx  
Fetch the item at position idx in the vector in src, and store it in dst. The idx value should be an unboxed unsigned 64-bit integer.

vector-ref/immediate s8:dst s8:src c8:idx  
Fill dst with the item idx elements into the vector at src. Useful for building data types using vectors.
vector-set! s8:dst s8:idx s8:src  [Instruction]
Store src into the vector dst at index idx. The idx value should be an unboxed unsigned 64-bit integer.

vector-set!/immediate s8:dst c8:idx s8:src  [Instruction]
Store src into the vector dst at index idx. Here idx is an immediate value.

struct-vtable s12:dst s12:src  [Instruction]
Store the vtable of src into dst.

allocate-struct s8:dst s8:vtable s8:nfields  [Instruction]
Allocate a new struct with vtable, and place it in dst. The struct will be constructed with space for nfields fields, which should correspond to the field count of the vtable. The idx value should be an unboxed unsigned 64-bit integer.

struct-ref s8:dst s8:src s8:idx  [Instruction]
Fetch the item at slot idx in the struct in src, and store it in dst. The idx value should be an unboxed unsigned 64-bit integer.

struct-set! s8:dst s8:idx s8:src  [Instruction]
Store src into the struct dst at slot idx. The idx value should be an unboxed unsigned 64-bit integer.

allocate-struct/immediate s8:dst s8:vtable c8:nfields  [Instruction]
structure-ref/immediate s8:dst s8:src c8:idx  [Instruction]
structure-set!/immediate s8:dst c8:idx s8:src  [Instruction]
Variants of the struct instructions, but in which the nfields or idx fields are immediate values.

class-of s12:dst s12:type  [Instruction]
Store the vtable of src into dst.

make-array s24:dst x8:_ s24:type x8:_ s24:fill x8:_ s24:bounds  [Instruction]
Make a new array with type, fill, and bounds, storing it in dst.

string-length s12:dst s12:src  [Instruction]
Store the length of the string in src in dst, as an unboxed unsigned 64-bit integer.

string-ref s8:dst s8:src s8:idx  [Instruction]
Fetch the character at position idx in the string in src, and store it in dst. The idx value should be an unboxed unsigned 64-bit integer.

string-set! s8:dst s8:idx s8:src  [Instruction]
Store the character src into the string dst at index idx. The idx value should be an unboxed unsigned 64-bit integer.

cons s8:dst s8:car s8:cdr  [Instruction]
Cons car and cdr, and store the result in dst.

car s12:dst s12:src  [Instruction]
Place the car of src in dst.
cdr s12:dst s12:src
   Place the cdr of src in dst.

set-car! s12:pair s12:car
   Set the car of dst to src.

set-cdr! s12:pair s12:cdr
   Set the cdr of dst to src.

Note that caddr and friends compile to a series of car and cdr instructions.

integer->char s12:dst s12:src
   Convert the u64 value in src to a Scheme character, and place it in dst.

char->integer s12:dst s12:src
   Convert the Scheme character in src to an integer, and place it in dst as an unboxed u64 value.

### 9.3.7.11 Inlined Atomic Instructions

See Section 6.22.4 [Atomics], page 441, for more on atomic operations in Guile.

make-atomic-box s12:dst s12:src
   Create a new atomic box initialized to src, and place it in dst.

atomic-box-ref s12:dst s12:box
   Fetch the value of the atomic box at box into dst.

atomic-box-set! s12:box s12:val
   Set the contents of the atomic box at box to val.

atomic-box-swap! s12:dst s12:box x8:_ s24:val
   Replace the contents of the atomic box at box to val and store the previous value at dst.

atomic-box-compare-and-swap! s12:dst s12:box x8:_ s24:expected x8:_ s24:desired
   If the value of the atomic box at box is the same as the SCM value at expected (in the sense of eq?), replace the contents of the box with the SCM value at desired. Otherwise does not update the box. Set dst to the previous value of the box in either case.

### 9.3.7.12 Inlined Mathematical Instructions

Inlining mathematical operations has the obvious advantage of handling fixnums without function calls or allocations. The trick, of course, is knowing when the result of an operation will be a fixnum, and there might be a couple bugs here.

More instructions could be added here over time.

All of these operations place their result in their first operand, dst.

add s8:dst s8:a s8:b
   Add a to b.
add/immediate s8:dst s8:src c8:imm

Add the unsigned integer imm to the value in src.

sub s8:dst s8:a s8:b

Subtract b from a.

sub/immediate s8:dst s8:src s8:imm

Subtract the unsigned integer imm from the value in src.

mul s8:dst s8:a s8:b

Multiply a and b.

div s8:dst s8:a s8:b

Divide a by b.

quo s8:dst s8:a s8:b

Divide a by b.

rem s8:dst s8:a s8:b

Divide a by b.

mod s8:dst s8:a s8:b

Compute the modulo of a by b.

ash s8:dst s8:a s8:b

Shift a arithmetically by b bits.

logand s8:dst s8:a s8:b

Compute the bitwise and of a and b.

logior s8:dst s8:a s8:b

Compute the bitwise inclusive or of a with b.

logxor s8:dst s8:a s8:b

Compute the bitwise exclusive or of a with b.

logsub s8:dst s8:a s8:b

Place the bitwise and of a and the bitwise not of b into dst.

9.3.7.13 Inlined Bytevector Instructions

Bytevector operations correspond closely to what the current hardware can do, so it makes sense to inline them to VM instructions, providing a clear path for eventual native compilation. Without this, Scheme programs would need other primitives for accessing raw bytes – but these primitives are as good as any.

bv-length s12:dst s12:src

Store the length of the bytevector in src in dst, as an unboxed unsigned 64-bit integer.
Fetch the item at byte offset \texttt{idx} in the bytevector \texttt{src}, and store it in \texttt{dst}. All accesses use native endianness.

The \texttt{idx} value should be an unboxed unsigned 64-bit integer.

The results are all written to the stack as unboxed values, either as signed 64-bit integers, unsigned 64-bit integers, or IEEE double floating point numbers.

Store \texttt{src} into the bytevector \texttt{dst} at byte offset \texttt{idx}. Multibyte values are written using native endianness.

The \texttt{idx} value should be an unboxed unsigned 64-bit integer.

The \texttt{src} values are all unboxed, either as signed 64-bit integers, unsigned 64-bit integers, or IEEE double floating point numbers.

### 9.3.7.14 Unboxed Integer Arithmetic

Guile supports two kinds of unboxed integers: unsigned 64-bit integers, and signed 64-bit integers. Guile prefers unsigned integers, in the sense that Guile’s compiler supports them better and the virtual machine has more operations that work on them. Still, signed integers are supported at least to allow \texttt{bv-s64-ref} and related instructions to avoid boxing their values.

Unbox the SCM value at \texttt{src} to a unsigned 64-bit integer, placing the result in \texttt{dst}. If the \texttt{src} value is not an exact integer in the unsigned 64-bit range, signal an error.

Box the unsigned 64-bit integer at \texttt{src} to a SCM value and place the result in \texttt{dst}. The result will be a fixnum or a bignum.
Chapter 9: Guile Implementation

load-u64 s24:dst au32:high-bits au32:low-bits

Load a 64-bit value formed by joining high-bits and low-bits, and write it to dst.

scm->s64 s12:dst s12:src
s64->scm s12:dst s12:src
load-s64 s24:dst as32:high-bits as32:low-bits

Like scm->u64, u64->scm, and load-u64, but for signed 64-bit integers.

Sometimes the compiler can know that we will only need a subset of the bits in an integer. In that case we can sometimes unbox an integer even if it might be out of range.

scm->u64/truncate s12:dst s12:src

Take the SCM value in dst and logand it with (1- (ash 1 64)). Place the unboxed result in dst.

br-if-u64-= s24:a x8:_ s24:b b1:invert x7:_ l24:offset
br-if-u64<- s24:a x8:_ s24:b b1:invert x7:_ l24:offset
br-if-u64-<= s24:a x8:_ s24:b b1:invert x7:_ l24:offset

If the unboxed unsigned 64-bit integer value in a is =, <, or <= to the unboxed unsigned 64-bit integer value in b, respectively, add offset to the current instruction pointer.

br-if-u64-=scm s24:a x8:_ s24:b b1:invert x7:_ l24:offset
br-if-u64<-scm s24:a x8:_ s24:b b1:invert x7:_ l24:offset
br-if-u64-<=scm s24:a x8:_ s24:b b1:invert x7:_ l24:offset

If the unboxed unsigned 64-bit integer value in a is =, <, or <= to the SCM value in b, respectively, add offset to the current instruction pointer.

uadd s8:dst s8:a s8:b
usub s8:dst s8:a s8:b
umul s8:dst s8:a s8:b

Like add, sub, and mul, except taking the operands as unboxed unsigned 64-bit integers, and producing the same. The result will be silently truncated to 64 bits.

uadd/immediate s8:dst s8:a c8:b
usub/immediate s8:dst s8:a c8:b
umul/immediate s8:dst s8:a c8:b

Like uadd, usub, and umul, except the second operand is an immediate unsigned 8-bit integer.

ulogand s8:dst s8:a s8:b
ulogior s8:dst s8:a s8:b
ulogxor s8:dst s8:a s8:b
ulogsub s8:dst s8:a s8:b

Like logand, logior, logxor, and logsub, but operating on unboxed unsigned 64-bit integers.

ulsh s8:dst s8:a s8:b

Shift the unboxed unsigned 64-bit integer in a left by b bits, also an unboxed unsigned 64-bit integer. Truncate to 64 bits and write to dst as an unboxed value. Only the lower 6 bits of b are used.
ursh \texttt{s8:dst s8:a s8:b} \quad \text{[Instruction]}

Like ulsh, but shifting right.

ulsh/immediate \texttt{s8:dst s8:a c8:b} \quad \text{[Instruction]}

Like ulsh and ursh, but encoding \texttt{b} as an immediate 8-bit unsigned integer.

\textbf{9.3.7.15 Unboxed Floating-Point Arithmetic}

\texttt{scm->f64 s12:dst s12:src} \quad \text{[Instruction]}

Unbox the SCM value at \texttt{src} to an IEEE double, placing the result in \texttt{dst}. If the \texttt{src} value is not a real number, signal an error.

\texttt{f64->scm s12:dst s12:src} \quad \text{[Instruction]}

Box the IEEE double at \texttt{src} to a SCM value and place the result in \texttt{dst}.

\texttt{load-f64 s24:dst au32:high-bits au32:low-bits} \quad \text{[Instruction]}

Load a 64-bit value formed by joining \texttt{high-bits} and \texttt{low-bits}, and write it to \texttt{dst}.

\texttt{fadd s8:dst s8:a s8:b} \quad \text{[Instruction]}

\texttt{fsub s8:dst s8:a s8:b} \quad \text{[Instruction]}

\texttt{fmul s8:dst s8:a s8:b} \quad \text{[Instruction]}

\texttt{fdiv s8:dst s8:a s8:b} \quad \text{[Instruction]}

Like \texttt{add}, \texttt{sub}, \texttt{div}, and \texttt{mul}, except taking the operands as unboxed IEEE double floating-point numbers, and producing the same.

\texttt{br-if-f64-=} \texttt{s24:a x8:_ s24:b b1:invert x7:_ l24:offset} \quad \text{[Instruction]}

\texttt{br-if-f64<-} \texttt{s24:a x8:_ s24:b b1:invert x7:_ l24:offset} \quad \text{[Instruction]}

\texttt{br-if-f64-<=} \texttt{s24:a x8:_ s24:b b1:invert x7:_ l24:offset} \quad \text{[Instruction]}

\texttt{br-if-f64->} \texttt{s24:a x8:_ s24:b b1:invert x7:_ l24:offset} \quad \text{[Instruction]}

\texttt{br-if-f64->=} \texttt{s24:a x8:_ s24:b b1:invert x7:_ l24:offset} \quad \text{[Instruction]}

If the unboxed IEEE double value in \texttt{a} is \texttt{=} \texttt{,} \texttt{<} \texttt{,} \texttt{<=} \texttt{,} \texttt{>} \texttt{,} \texttt{or} \texttt{=} \texttt{to the unboxed IEEE double value in \texttt{b}, respectively, add \texttt{offset} to the current instruction pointer.}

\textbf{9.4 Compiling to the Virtual Machine}

Compilers! The word itself inspires excitement and awe, even among experienced practitioners. But a compiler is just a program: an eminently hackable thing. This section aims to describe Guile’s compiler in such a way that interested Scheme hackers can feel comfortable reading and extending it.

See Section 6.18 [Read/Load/Eval/Compile], page 379, if you’re lost and you just wanted to know how to compile your \texttt{.scm} file.

\textbf{9.4.1 Compiler Tower}

Guile’s compiler is quite simple – it’s \textit{compilers}, to put it more accurately. Guile defines a tower of languages, starting at Scheme and progressively simplifying down to languages that resemble the VM instruction set (see Section 9.3.7 [Instruction Set], page 819).

Each language knows how to compile to the next, so each step is simple and understandable. Furthermore, this set of languages is not hardcoded into Guile, so it is possible for the user to add new high-level languages, new passes, or even different compilation targets.
Languages are registered in the module, (system base language):

(use-modules (system base language))

They are registered with the define-language form.

\[
define-language [ #:name ] [ #:title ] [ #:reader ] [ #:printer ] [ #:parser=#f ] [ #:compilers=(') ] [ #:decompilers=(') ] [ #:evaluator=#f ] [ #:joiner=#f ] [ #:for-humans?=#t ] [ #:make-default-environment=make-fresh-user-module ]
\]

Define a language.

This syntax defines a <language> object, bound to name in the current environment. In addition, the language will be added to the global language set. For example, this is the language definition for Scheme:

\[
(define-language scheme
  #:title "Scheme"
  #:reader (lambda (port env) ...)
  #:compilers '((tree-il . ,compile-tree-il))
  #:decompilers '((tree-il . ,decompile-tree-il))
  #:evaluator (lambda (x module) (primitive-eval x))
  #:printer write
  #:make-default-environment (lambda () ...))
\]

The interesting thing about having languages defined this way is that they present a uniform interface to the read-eval-print loop. This allows the user to change the current language of the REPL:

\[
scheme@(guile-user)> ,language tree-il
Happy hacking with Tree Intermediate Language! To switch back, type ',L scheme'.
tree-il@(guile-user)> ,L scheme
Happy hacking with Scheme! To switch back, type ',L tree-il'.
scheme@(guile-user)>
\]

Languages can be looked up by name, as they were above.

\[
lookup-language name
\]

Looks up a language named name, autoloading it if necessary.

Languages are autoloaded by looking for a variable named name in a module named (language name spec).

The language object will be returned, or #f if there does not exist a language with that name.

Defining languages this way allows us to programmatically determine the necessary steps for compiling code from one language to another.

\[
lookup-compilation-order from to
\]

Recursively traverses the set of languages to which from can compile, depth-first, and return the first path that can transform from to to. Returns #f if no path is found.

This function memoizes its results in a cache that is invalidated by subsequent calls to define-language, so it should be quite fast.
There is a notion of a “current language”, which is maintained in the `current-language` parameter, defined in the core `(guile)` module. This language is normally Scheme, and may be rebound by the user. The run-time compilation interfaces (see Section 6.18 [Read/Load/Eval/Compile], page 379) also allow you to choose other source and target languages.

The normal tower of languages when compiling Scheme goes like this:

- Scheme
- Tree Intermediate Language (Tree-IL)
- Continuation-Passing Style (CPS)
- Bytecode

As discussed before (see Section 9.3.6 [Object File Format], page 817), bytecode is in ELF format, ready to be serialized to disk. But when compiling Scheme at run time, you want a Scheme value: for example, a compiled procedure. For this reason, so as not to break the abstraction, Guile defines a fake language at the bottom of the tower:

- Value

Compiling to `value` loads the bytecode into a procedure, turning cold bytes into warm code.

Perhaps this strangeness can be explained by example: `compile-file` defaults to compiling to bytecode, because it produces object code that has to live in the barren world outside the Guile runtime; but `compile` defaults to compiling to `value`, as its product re-enters the Guile world.

Indeed, the process of compilation can circulate through these different worlds indefinitely, as shown by the following quine:

```
((lambda (x) ((compile x) x)) '(lambda (x) ((compile x) x)))
```

### 9.4.2 The Scheme Compiler

The job of the Scheme compiler is to expand all macros and all of Scheme to its most primitive expressions. The definition of “primitive expression” is given by the inventory of constructs provided by Tree-IL, the target language of the Scheme compiler: procedure calls, conditionals, lexical references, and so on. This is described more fully in the next section.

The tricky and amusing thing about the Scheme-to-Tree-IL compiler is that it is completely implemented by the macro expander. Since the macro expander has to run over all of the source code already in order to expand macros, it might as well do the analysis at the same time, producing Tree-IL expressions directly.

Because this compiler is actually the macro expander, it is extensible. Any macro which the user writes becomes part of the compiler.

The Scheme-to-Tree-IL expander may be invoked using the generic `compile` procedure:

```
(compile '(+ 1 2) #:from 'scheme #:to 'tree-il)
⇒ #<tree-il (call (toplevel +) (const 1) (const 2))>
```

```
(compile foo #:from 'scheme #:to 'tree-il)
```

is entirely equivalent to calling the macro expander as `(macroexpand foo 'c '(compile load eval))`. See Section 6.10.9
[Macro Expansion], page 280. compile-tree-il, the procedure dispatched by compile to 'tree-il, is a small wrapper around macroexpand, to make it conform to the general form of compiler procedures in Guile’s language tower.

Compiler procedures take three arguments: an expression, an environment, and a keyword list of options. They return three values: the compiled expression, the corresponding environment for the target language, and a “continuation environment”. The compiled expression and environment will serve as input to the next language’s compiler. The “continuation environment” can be used to compile another expression from the same source language within the same module.

For example, you might compile the expression, (define-module (foo)). This will result in a Tree-IL expression and environment. But if you compiled a second expression, you would want to take into account the compile-time effect of compiling the previous expression, which puts the user in the (foo) module. That is the purpose of the “continuation environment”; you would pass it as the environment when compiling the subsequent expression.

For Scheme, an environment is a module. By default, the compile and compile-file procedures compile in a fresh module, such that bindings and macros introduced by the expression being compiled are isolated:

```
(eq? (current-module) (compile '(current-module)))
⇒ #f
```

```
(compile '(define hello 'world))
(defined? 'hello)
⇒ #f
```

```
(define / *)
(eq? (compile '/) /)
⇒ #f
```

Similarly, changes to the current-reader fluid (see Section 6.18.6 [Loading], page 388) are isolated:

```
(compile '(fluid-set! current-reader (lambda args 'fail)))
(fluid-ref current-reader)
⇒ #f
```

Nevertheless, having the compiler and compilee share the same name space can be achieved by explicitly passing (current-module) as the compilation environment:

```
(define hello 'world)
(compile 'hello #:env (current-module))
⇒ world
```

### 9.4.3 Tree-IL

Tree Intermediate Language (Tree-IL) is a structured intermediate language that is close in expressive power to Scheme. It is an expanded, pre-analyzed Scheme.

Tree-IL is “structured” in the sense that its representation is based on records, not S-expressions. This gives a rigidity to the language that ensures that compiling to a lower-level language only requires a limited set of transformations. For example, the Tree-IL
type `<const>` is a record type with two fields, `src` and `exp`. Instances of this type are created via `make-const`. Fields of this type are accessed via the `const-src` and `const-exp` procedures. There is also a predicate, `const?`. See Section 6.6.17 [Records], page 221, for more information on records.

All Tree-IL types have a `src` slot, which holds source location information for the expression. This information, if present, will be residualized into the compiled object code, allowing backtraces to show source information. The format of `src` is the same as that returned by Guile’s `source-properties` function. See Section 6.26.2 [Source Properties], page 471, for more information.

Although Tree-IL objects are represented internally using records, there is also an equivalent S-expression external representation for each kind of Tree-IL. For example, the S-expression representation of `#<const src: #f exp: 3>` expression would be:

```
(const 3)
```

Users may program with this format directly at the REPL:

```
scheme@(guile-user)> ,language tree-il
Happy hacking with Tree Intermediate Language! To switch back, type ',L scheme'.
tree-il@(guile-user)> (call (primitive +) (const 32) (const 10))
⇒ 42
```

The `src` fields are left out of the external representation.

One may create Tree-IL objects from their external representations via calling `parse-tree-il`, the reader for Tree-IL. If any source information is attached to the input S-expression, it will be propagated to the resulting Tree-IL expressions. This is probably the easiest way to compile to Tree-IL: just make the appropriate external representations in S-expression format, and let `parse-tree-il` take care of the rest.

[^void]: [Scheme Variable] [External Representation]

An empty expression. In practice, equivalent to Scheme's `(if #f #f)`.

[^const]: [Scheme Variable] [External Representation]

A constant.

[^primitive-ref]: [Scheme Variable] [External Representation]

A reference to a “primitive”. A primitive is a procedure that, when compiled, may be open-coded. For example, `cons` is usually recognized as a primitive, so that it compiles down to a single instruction.

Compilation of Tree-IL usually begins with a pass that resolves some `<module-ref>` and `<toplevel-ref>` expressions to `<primitive-ref>` expressions. The actual compilation pass has special cases for calls to certain primitives, like `apply` or `cons`.

[^lexical-ref]: [Scheme Variable] [External Representation]

A reference to a lexically-bound variable. The `name` is the original name of the variable in the source program. `gensym` is a unique identifier for this variable.
<lexical-set>  src name gensym exp  
(set! (lexical name gensym) exp)  
Sets a lexically-bound variable.

<module-ref>  src mod name public?  
(@ mod name)  
(@@ mod name)  
A reference to a variable in a specific module. mod should be the name of the module, e.g. (guile-user).
If public? is true, the variable named name will be looked up in mod’s public interface, and serialized with @; otherwise it will be looked up among the module’s private bindings, and is serialized with @@.

<module-set>  src mod name public? exp  
(set! (@ mod name) exp)  
(set! (@@ mod name) exp)  
Sets a variable in a specific module.

<toplevel-ref>  src name  
(toplevel name)  
References a variable from the current procedure’s module.

<toplevel-set>  src name exp  
(set! (toplevel name) exp)  
Sets a variable in the current procedure’s module.

<toplevel-define>  src name exp  
(define name exp)  
Defines a new top-level variable in the current procedure’s module.

<conditional>  src test then else  
(if test then else)  
A conditional. Note that else is not optional.

<call>  src proc args  
(call proc . args)  
A procedure call.

<primcall>  src name args  
(primcall name . args)  
A call to a primitive. Equivalent to (call (primitive name) . args). This construct is often more convenient to generate and analyze than <call>.
As part of the compilation process, instances of (call (primitive name) . args) are transformed into primcalls.

<seq>  src head tail  
(seq head tail)  
A sequence. The semantics is that head is evaluated first, and any resulting values are ignored. Then tail is evaluated, in tail position.
A closure. \textit{meta} is an association list of properties for the procedure. \textit{body} is a single Tree-IL expression of type \texttt{<lambda-case>}. As the \texttt{<lambda-case>} clause can chain to an alternate clause, this makes Tree-IL's \texttt{<lambda>} have the expressiveness of Scheme's \texttt{case-lambda}.

One clause of a case-lambda. A lambda expression in Scheme is treated as a case-lambda with one clause.

\textit{req} is a list of the procedure's required arguments, as symbols. \textit{opt} is a list of the optional arguments, or \#f if there are no optional arguments. \textit{rest} is the name of the rest argument, or \#f.

\textit{kw} is a list of the form, \texttt{(allow-other-keys? (keyword name var) \ldots)}, where keyword is the keyword corresponding to the argument named \textit{name}, and whose corresponding gensym is \textit{var}. \textit{inits} are tree-il expressions corresponding to all of the optional and keyword arguments, evaluated to bind variables whose value is not supplied by the procedure caller. Each \textit{init} expression is evaluated in the lexical context of previously bound variables, from left to right.

\textit{gensyms} is a list of gensyms corresponding to all arguments: first all of the required arguments, then the optional arguments if any, then the rest argument if any, then all of the keyword arguments.

\textit{body} is the body of the clause. If the procedure is called with an appropriate number of arguments, \textit{body} is evaluated in tail position. Otherwise, if there is an alternate, it should be a \texttt{<lambda-case>} expression, representing the next clause to try. If there is no alternate, a wrong-number-of-arguments error is signaled.

Lexical binding, like Scheme's \texttt{let}. \textit{names} are the original binding names, \textit{gensyms} are gensyms corresponding to the \textit{names}, and \textit{vals} are Tree-IL expressions for the values. \textit{exp} is a single Tree-IL expression.

A version of \texttt{<let>} that creates recursive bindings, like Scheme's \texttt{letrec}, or \texttt{letrec*} if in-order? is true.

A dynamic prompt. Instates a prompt named \textit{tag}, an expression, during the dynamic extent of the execution of \textit{body}, also an expression. If an abort occurs to this prompt, control will be passed to \textit{handler}, also an expression, which should be a procedure. The first argument to the handler procedure will be the captured continuation, followed
by all of the values passed to the abort. If escape-only? is true, the handler should be a \(<\text{lambda}>\) with a single \(<\text{lambda-case}>\) body expression with no optional or keyword arguments, and no alternate, and whose first argument is unreferenced. See Section 6.13.5 [Prompts], page 303, for more information.

\(<\text{abort}>\)  \textit{tag args tail}  \quad \text{[Scheme Variable]}  \\
\text{(abort tag args tail)} \quad \text{[External Representation]}  \\
An abort to the nearest prompt with the name \textit{tag}, an expression. \textit{args} should be a list of expressions to pass to the prompt’s handler, and \textit{tail} should be an expression that will evaluate to a list of additional arguments. An abort will save the partial continuation, which may later be reinstated, resulting in the \(<\text{abort}>\) expression evaluating to some number of values.

There are two Tree-IL constructs that are not normally produced by higher-level compilers, but instead are generated during the source-to-source optimization and analysis passes that the Tree-IL compiler does. Users should not generate these expressions directly, unless they feel very clever, as the default analysis pass will generate them as necessary.

\(<\text{let-values}>\)  \textit{src names gensyms exp body}  \quad \text{[Scheme Variable]}  \\
\text{(let-values names gensyms exp body)} \quad \text{[External Representation]}  \\
Like Scheme’s \textit{receive} – binds the values returned by evaluating \textit{exp} to the \textit{lambda}-like bindings described by \textit{gensyms}. That is to say, \textit{gensyms} may be an improper list.

\(<\text{let-values}>\) is an optimization of a \(<\text{call}>\) to the primitive, \textit{call-with-values}.

\(<\text{fix}>\)  \textit{src names gensyms vals body}  \quad \text{[Scheme Variable]}  \\
\text{(fix names gensyms vals body)} \quad \text{[External Representation]}  \\
Like \(<\text{letrec}>\), but only for \textit{vals} that are unset \textit{lambda} expressions.

\textit{fix} is an optimization of \textit{letrec} (and \textit{let}).

Tree-IL is a convenient compilation target from source languages. It can be convenient as a medium for optimization, though CPS is usually better. The strength of Tree-IL is that it does not fix order of evaluation, so it makes some code motion a bit easier.

Optimization passes performed on Tree-IL currently include:

- Open-coding (turning toplevel-refs into primitive-refs, and calls to primitives to prim-calls)
- Partial evaluation (comprising inlining, copy propagation, and constant folding)

### 9.4.4 Continuation-Passing Style

Continuation-passing style (CPS) is Guile’s principal intermediate language, bridging the gap between languages for people and languages for machines. CPS gives a name to every part of a program: every control point, and every intermediate value. This makes it an excellent medium for reasoning about programs, which is the principal job of a compiler.

#### 9.4.4.1 An Introduction to CPS

Consider the following Scheme expression:

\(\text{(begin} \ldots\))
(display "The sum of 32 and 10 is: ")
(display 42)
(newline))

Let us identify all of the sub-expressions in this expression, annotating them with unique labels:

(begin
 (display "The sum of 32 and 10 is: ")
 |k1 k2
 k0
 (display 42)
 |k4 k5
 k3
 (newline))
 |k7
 k6

Each of these labels identifies a point in a program. One label may be the continuation of another label. For example, the continuation of k7 is k6. This is because after evaluating the value of newline, performed by the expression labelled k7, we continue to apply it in k6.

Which expression has k0 as its continuation? It is either the expression labelled k1 or the expression labelled k2. Scheme does not have a fixed order of evaluation of arguments, though it does guarantee that they are evaluated in some order. Unlike general Scheme, continuation-passing style makes evaluation order explicit. In Guile, this choice is made by the higher-level language compilers.

Let us assume a left-to-right evaluation order. In that case the continuation of k1 is k2, and the continuation of k2 is k0.

With this example established, we are ready to give an example of CPS in Scheme:

(l lambda (ktail)
 (let ((k1 (lambda ()
 (let ((k2 (lambda (proc)
 (let ((k0 (lambda (arg0)
 (proc k4 arg0))))
 (k0 "The sum of 32 and 10 is: ")))))))
 (k2 display)))))
 (k4 (lambda _
 (let ((k5 (lambda (proc)
 (let ((k3 (lambda (arg0)
 (proc k7 arg0))))
 (k3 42))))))
 (k5 display)))))
 (k7 (lambda _
 (let ((k6 (lambda (proc)
 (proc ktail))))
 (k6 newline))))))
 (k1))

Holy code explosion, Batman! What’s with all the lambdas? Indeed, CPS is by nature much more verbose than “direct-style” intermediate languages like Tree-IL. At the same time, CPS is simpler than full Scheme, because it makes things more explicit.
In the original program, the expression labelled k0 is in effect context. Any values it returns are ignored. In Scheme, this fact is implicit. In CPS, we can see it explicitly by noting that its continuation, k4, takes any number of values and ignores them. Compare this to k2, which takes a single value; in this way we can say that k1 is in a “value” context. Likewise k6 is in tail context with respect to the expression as a whole, because its continuation is the tail continuation, ktail. CPS makes these details manifest, and gives them names.

9.4.4.2 CPS in Guile

Guile’s CPS language is composed of continuations. A continuation is a labelled program point. If you are used to traditional compilers, think of a continuation as a trivial basic block. A program is a “soup” of continuations, represented as a map from labels to continuations.

Like basic blocks, each continuation belongs to only one function. Some continuations are special, like the continuation corresponding to a function’s entry point, or the continuation that represents the tail of a function. Others contain a term. A term contains an expression, which evaluates to zero or more values. The term also records the continuation to which it will pass its values. Some terms, like conditional branches, may continue to one of a number of continuations.

Continuation labels are small integers. This makes it easy to sort them and to group them into sets. Whenever a term refers to a continuation, it does so by name, simply recording the label of the continuation. Continuation labels are unique among the set of labels in a program.

Variables are also named by small integers. Variable names are unique among the set of variables in a program.

For example, a simple continuation that receives two values and adds them together can be matched like this, using the match form from (ice-9 match):

```
(match cont
  ((#kargs (x-name y-name) (x-var y-var)
      (#continue k src (#primcall '+ (x-var y-var))))
   (format #t "Add ~a and ~a and pass the result to label ~a"
          x-var y-var k))
```

Here we see the most common kind of continuation, $kargs, which binds some number of values to variables and then evaluates a term.

$\text{kargs names vars term}$ [CPS Continuation]

Bind the incoming values to the variables vars, with original names names, and then evaluate term.

The names of a $kargs$ are just for debugging, and will end up residualized in the object file for use by the debugger.

The term in a $kargs$ is always a $continue$, which evaluates an expression and continues to a continuation.

$\text{continue k src exp}$ [CPS Term]

Evaluate the expression exp and pass the resulting values (if any) to the continuation labelled k. The source information associated with the expression may be found in src, which is either an alist as in source-properties or is #f if there is no associated source.
There are a number of expression kinds. Above you see an example of \$primcall.

\$primcall name args

Perform the primitive operation identified by name, a well-known symbol, passing it the arguments args, and pass all resulting values to the continuation. The set of available primitives includes all primitives known to Tree-IL and then some more; see the source code for details.

The variables that are used by \$primcall, or indeed by any expression, must be defined before the expression is evaluated. An equivalent way of saying this is that predecessor $kargs continuation(s) that bind the variables(s) used by the expression must dominate the continuation that uses the expression: definitions dominate uses. This condition is trivially satisfied in our example above, but in general to determine the set of variables that are in “scope” for a given term, you need to do a flow analysis to see what continuations dominate a term. The variables that are in scope are those variables defined by the continuations that dominate a term.

Here is an inventory of the kinds of expressions in Guile’s CPS language, besides \$primcall which has already been described. Recall that all expressions are wrapped in a $continue term which specifies their continuation.

\$const val

Continue with the constant value val.

\$prim name

Continue with the procedure that implements the primitive operation named by name.

\$call proc args

Call proc with the arguments args, and pass all values to the continuation. proc and the elements of the args list should all be variable names. The continuation identified by the term’s k should be a $kreceive or a $ktail instance.

\$values args

Pass the values named by the list args to the continuation.

\$branch kt exp

Evaluate the branching expression exp, and continue to kt with zero values if the test evaluates to true. Otherwise continue to the continuation named in the outer $continue term.

Only certain expressions are valid in a \$branch. Compiling a \$branch avoids allocating space for the test variable, so the expression should be evaluable without temporary values. In practice this condition is true for \$primcalls to null?, =, and similar primitives that have corresponding br-if-foo VM operations; see the source code for full details. When in doubt, bind the test expression to a variable, and branch on a \$values expression that references that variable. The optimizer should inline the reference if possible.

\$prompt escape? tag handler

Push a prompt on the stack identified by the variable name tag, which may be escape-only if escape? is true, and continue with zero values. If the body aborts to this prompt, control will proceed at the continuation labelled handler, which should be a $kreceive continuation. Prompts are later popped by pop-prompt primcalls.
There are two sub-languages of CPS, higher-order CPS and first-order CPS. The difference is that in higher-order CPS, there are $fun$ and $rec$ expressions that bind functions or mutually-recursive functions in the implicit scope of their use sites. Guile transforms higher-order CPS into first-order CPS by closure conversion, which chooses representations for all closures and which arranges to access free variables through the implicit closure parameter that is passed to every function call.

$fun \ body$ [CPS Expression]
Continue with a procedure. $body$ names the entry point of the function, which should be a $kfun$. This expression kind is only valid in higher-order CPS, which is the CPS language before closure conversion.

$rec \ names \ vars \ funs$ [CPS Expression]
Continue with a set of mutually recursive procedures denoted by $names$, $vars$, and $funs$. $names$ is a list of symbols, $vars$ is a list of variable names (unique integers), and $funs$ is a list of $fun$ values. Note that the $kargs$ continuation should also define $names/vars$ bindings.

The contification pass will attempt to transform the functions declared in a $rec$ into local continuations. Any remaining $fun$ instances are later removed by the closure conversion pass. By default, a closure is represented as an object built by a $closure$ expression.

$closure \ label \ nfree$ [CPS Expression]
Build a closure that joins the code at the continuation named $label$ with space for $nfree$ free variables. The variables will be initialized later via free-set! primcalls. This expression kind is part of first-order CPS.

If the closure can be proven to never escape its scope then other lighter-weight representations can be chosen. Additionally, if all call sites are known, closure conversion will hard-wire the calls by lowering $call$ to $callk$.

$callk \ label \ proc \ args$ [CPS Expression]
Like $call$, but for the case where the call target is known to be in the same compilation unit. $label$ should denote some $kfun$ continuation in the program. In this case the $proc$ is simply an additional argument, since it is not used to determine the call target at run-time.

At this point we have described terms, expressions, and the most common kind of continuation, $kargs$. $kargs$ is used when the predecessors of the continuation can be instructed to pass the values where the continuation wants them. For example, if a $kargs$ continuation $k$ binds a variable $v$, and the compiler decides to allocate $v$ to slot 6, all predecessors of $k$ should put the value for $v$ in slot 6 before jumping to $k$. One situation in which this isn’t possible is receiving values from function calls. Guile has a calling convention for functions which currently places return values on the stack. A continuation of a call must check that the number of values returned from a function matches the expected number of values, and then must shuffle or collect those values to named variables. $kreceive$ denotes this kind of continuation.
$kreceive\ arity\ k$ [CPS Continuation]
Receive values on the stack. Parse them according to \textit{arity}, and then proceed with
the parsed values to the $\$kargs$ continuation labelled \textit{k}. As a limitation specific to
$kreceive, arity$ may only contain required and rest arguments.

$arity$ is a helper data structure used by $kreceive$ and also by $kclause$, described
below.

$arity\ req\ opt\ rest\ kw\ allow\-other\-keys?$ [CPS Data]
A data type declaring an arity. \textit{req} and \textit{opt} are lists of source names of required and
optional arguments, respectively. \textit{rest} is either the source name of the rest variable, or
#f if this arity does not accept additional values. \textit{kw} is a list of the form ((\textit{keyword
name var}) ...), describing the keyword arguments. \textit{allow\-other\-keys?} is true if other
keyword arguments are allowed and false otherwise.

Note that all of these names with the exception of the \textit{vars} in the \textit{kw} list are source
names, not unique variable names.

Additionally, there are three specific kinds of continuations that are only used in function
entries.

$kfun\ src\ meta\ self\ tail\ clause$ [CPS Continuation]
Declare a function entry. \textit{src} is the source information for the procedure declaration,
and \textit{meta} is the metadata alist as described above in Tree-IL’s <lambda>. \textit{self} is
a variable bound to the procedure being called, and which may be used for self-
references. \textit{tail} is the label of the $k\tail$ for this function, corresponding to the
function’s tail continuation. \textit{clause} is the label of the first $k\clause$ for the first
\textit{case-lambda} clause in the function, or otherwise #f.

$ktail$
A tail continuation.

$kclause\ arity\ cont\ alternate$ [CPS Continuation]
A clause of a function with a given arity. Applications of a function with a compatible
set of actual arguments will continue to the continuation labelled \textit{cont}, a $k\args$
instance representing the clause body. If the arguments are incompatible, control
proceeds to \textit{alternate}, which is a $k\clause$ for the next clause, or #f if there is no
next clause.

9.4.4.3 Building CPS
Unlike Tree-IL, the CPS language is built to be constructed and deconstructed with ab-
stract macros instead of via procedural constructors or accessors, or instead of S-expression
matching.

Deconstruction and matching is handled adequately by the match form from (ice-9
match). See Section 7.7 [Pattern Matching], page 691. Construction is handled by a set of
mutually builder macros: build-term, build-cont, and build-exp.

In the following interface definitions, consider \textit{term} and \textit{exp} to be built by build-term or
build-exp, respectively. Consider any other name to be evaluated as a Scheme expression.
Many of these forms recognize unquote in some contexts, to splice in a previously-built
value; see the specifications below for full details.
build-term, val
build-term ($continue k src exp)
build-exp, val
build-exp ($const val)
build-exp ($prim name)
build-exp ($branch kt exp)
build-exp ($fun kentry)
build-exp ($rec names syms funs)
build-exp ($closure k nfree)
build-exp ($call proc (arg ...))
build-exp ($call proc args)
build-exp ($callk k proc (arg ...))
build-exp ($callk k proc args)
build-exp ($primcall name (arg ...))
build-exp ($primcall name args)
build-exp ($values (arg ...))
build-exp ($values args)
build-exp ($prompt escape? tag handler)
build-cont, val
build-cont ($kargs (name ...) (sym ...) term)
build-cont ($kargs names syms term)
build-cont ($kreceive req rest kargs)
build-cont ($kfun src meta self ktail kclause)
build-cont ($kclause ,arity kbody kalt)
build-cont ($kclause (req opt rest kw aok?) kbody)

Construct a CPS term, expression, or continuation.

There are a few more miscellaneous interfaces as well.

make-arity req opt rest kw allow-other-keywords?

A procedural constructor for $arity objects.

rewrite-term val (pat term) ...
rewrite-exp val (pat exp) ...
rewrite-cont val (pat cont) ...

Match val against the series of patterns pat..., using match. The body of the matching clause should be a template in the syntax of build-term, build-exp, or build-cont, respectively.

9.4.4.4 CPS Soup

We describe programs in Guile’s CPS language as being a kind of “soup” because all continuations in the program are mixed into the same “pot”, so to speak, without explicit markers as to what function or scope a continuation is in. A program in CPS is a map from continuation labels to continuation values. As discussed in the introduction, a continuation label is an integer. No label may be negative.

As a matter of convention, label 0 should map to the $kfun continuation of the entry to the program, which should be a function of no arguments. The body of a function consists
of the labelled continuations that are reachable from the function entry. A program can refer to other functions, either via $\text{fun}$ and $\text{rec}$ in higher-order CPS, or via $\text{closure}$ and $\text{callk}$ in first-order CPS. The program logically contains all continuations of all functions reachable from the entry function. A compiler pass may leave unreachable continuations in a program; subsequent compiler passes should ensure that their transformations and analyses only take reachable continuations into account. It’s OK though if transformation runs over all continuations if including the unreachable continuations has no effect on the transformations on the live continuations.

The “soup” itself is implemented as an intmap, a functional array-mapped trie specialized for integer keys. Intmaps associate integers with values of any kind. Currently intmaps are a private data structure only used by the CPS phase of the compiler. To work with intmaps, load the $(\text{language cps intmap})$ module:

```
(use-modules (language cps intmap))
```

Intmaps are functional data structures, so there is no constructor as such: one can simply start with the empty intmap and add entries to it.

```
(intmap? empty-intmap) ⇒ #t
(define x (intmap-add empty-intmap 42 "hi"))
(intmap? x) ⇒ #t
(intmap-ref x 42) ⇒ "hi"
(intmap-ref x 43) ⇒ error: 43 not present
(intmap-ref x 43 (lambda (k) "yo!")) ⇒ "yo"
(intmap-add x 42 "hej") ⇒ error: 42 already present
```

intmap-ref and intmap-add are the core of the intmap interface. There is also intmap-replace, which replaces the value associated with a given key, requiring that the key was present already, and intmap-remove, which removes a key from an intmap.

Intmaps have a tree-like structure that is well-suited to set operations such as union and intersection, so there are also the binary intmap-union and intmap-intersect procedures. If the result is equivalent to either argument, that argument is returned as-is; in that way, one can detect whether the set operation produced a new result simply by checking with eq?. This makes intmaps useful when computing fixed points.

If a key is present in both intmaps and the associated values are not the same in the sense of eq?, the resulting value is determined by a “meet” procedure, which is the optional last argument to intmap-union, intmap-intersect, and also to intmap-add, intmap-replace, and similar functions. The meet procedure will be called with the two values and should return the intersected or unioned value in some domain-specific way. If no meet procedure is given, the default meet procedure will raise an error.

To traverse over the set of values in an intmap, there are the intmap-next and intmap-prev procedures. For example, if intmap x has one entry mapping 42 to some value, we would have:

```
(intmap-next x) ⇒ 42
(intmap-next x 0) ⇒ 42
(intmap-next x 42) ⇒ 42
(intmap-next x 43) ⇒ #f
(intmap-prev x) ⇒ 42
(intmap-prev x 42) ⇒ 42
```
There is also the intmap-fold procedure, which folds over keys and values in the intmap from lowest to highest value, and intmap-fold-right which does so in the opposite direction. These procedures may take up to 3 seed values. The number of values that the fold procedure returns is the number of seed values.

\[
(\text{define } q \equiv (\text{intmap-add} (\text{intmap-add} \text{empty-intmap} 1 2) 3 4))
\]

\[
(\text{intmap-fold acons q '()} \Rightarrow ((3 . 4) (1 . 2))
\]

\[
(\text{intmap-fold-right acons q '()} \Rightarrow ((1 . 2) (3 . 4))
\]

When an entry in an intmap is updated (removed, added, or changed), a new intmap is created that shares structure with the original intmap. This operation ensures that the result of existing computations is not affected by future computations: no mutation is ever visible to user code. This is a great property in a compiler data structure, as it lets us hold a copy of a program before a transformation and use it while we build a post-transformation program. Updating an intmap is O(log \(n\)) in the size of the intmap.

However, the O(log \(n\)) allocation costs are sometimes too much, especially in cases when we know that we can just update the intmap in place. As an example, say we have an intmap mapping the integers 1 to 100 to the integers 42 to 141. Let’s say that we want to transform this map by adding 1 to each value. There is already an efficient intmap-map procedure in the (language cps utils) module, but if we didn’t know about that we might do:

\[
(\text{define } (\text{intmap-increment} \text{map})
\]

\[
(\text{let lp ((k 0) (map map))}
\]

\[
(\text{let ((k (intmap-next map k)))}
\]

\[
(\text{if k}
\]

\[
(\text{let ((v (intmap-ref map k))})
\]

\[
(\text{lp (1+ k) (intmap-replace map k (1+ v))})
\]

\[
\text{map})
\]

Observe that the intermediate values created by intmap-replace are completely invisible to the program – only the last result of intmap-replace value is needed. The rest might as well share state with the last one, and we could update in place. Guile allows this kind of interface via transient intmaps, inspired by Clojure’s transient interface (http://clojure.org/transients).

The in-place intmap-add! and intmap-replace! procedures return transient intmaps. If one of these in-place procedures is called on a normal persistent intmap, a new transient intmap is created. This is an O(1) operation. In all other respects the interface is like their persistent counterparts, intmap-add and intmap-replace. If an in-place procedure is called on a transient intmap, the intmap is mutated in-place and the same value is returned.

If a persistent operation like intmap-add is called on a transient intmap, the transient’s mutable substructure is then marked as persistent, and intmap-add then runs on a new persistent intmap sharing structure but not state with the original transient. Mutating a transient will cause enough copying to ensure that it can make its change, but if part of its substructure is already “owned” by it, no more copying is needed.

We can use transients to make intmap-increment more efficient. The two changed elements have been marked like this.

\[
(\text{define } (\text{intmap-increment} \text{map})
\]
(let lp ((k 0) (map map))
  (let ((k (intmap-next map k)))
    (if k
      (let ((v (intmap-ref map k)))
        (lp (1+ k) (intmap-replace! map k (1+ v))))
      (persistent-intmap map))))

Be sure to tag the result as persistent using the `persistent-intmap` procedure to prevent the mutability from leaking to other parts of the program. For added paranoia, you could call `persistent-intmap` on the incoming map, to ensure that if it were already transient, that the mutations in the body of `intmap-increment` wouldn't affect the incoming value.

In summary, programs in CPS are intmaps whose values are continuations. See the source code of `(language cps utils)` for a number of useful facilities for working with CPS values.

### 9.4.4.5 Compiling CPS

Compiling CPS in Guile has three phases: conversion, optimization, and code generation.

CPS conversion is the process of taking a higher-level language and compiling it to CPS. Source languages can do this directly, or they can convert to Tree-IL (which is probably easier) and let Tree-IL convert to CPS later. Going through Tree-IL has the advantage of running Tree-IL optimization passes, like partial evaluation. Also, the compiler from Tree-IL to CPS handles assignment conversion, in which assigned local variables (in Tree-IL, locals that are `<lexical-set>`) are converted to being boxed values on the heap. See Section 9.3.4 [Variables and the VM], page 815.

After CPS conversion, Guile runs some optimization passes over the CPS. Most optimization in Guile is done on the CPS language. The one major exception is partial evaluation, which for historic reasons is done on Tree-IL.

The major optimization performed on CPS is contification, in which functions that are always called with the same continuation are incorporated directly into a function’s body. This opens up space for more optimizations, and turns procedure calls into `goto`. It can also make loops out of recursive function nests. Guile also does dead code elimination, common subexpression elimination, loop peeling and invariant code motion, and range and type inference.

The rest of the optimization passes are really cleanups and canonicalizations. CPS spans the gap between high-level languages and low-level bytecodes, which allows much of the compilation process to be expressed as source-to-source transformations. Such is the case for closure conversion, in which references to variables that are free in a function are converted to closure references, and in which functions are converted to closures. There are a few more passes to ensure that the only primcalls left in the term are those that have a corresponding instruction in the virtual machine, and that their continuations expect the right number of values.

Finally, the backend of the CPS compiler emits bytecode for each function, one by one. To do so, it determines the set of live variables at all points in the function. Using this liveness information, it allocates stack slots to each variable, such that a variable can live in one slot for the duration of its lifetime, without shuffling. (Of course, variables with
disjoint lifetimes can share a slot.) Finally the backend emits code, typically just one VM
instruction, for each continuation in the function.

9.4.5 Bytecode

As mentioned before, Guile compiles all code to bytecode, and that bytecode is contained
in ELF images. See Section 9.3.6 [Object File Format], page 817, for more on Guile’s use
of ELF.

To produce a bytecode image, Guile provides an assembler and a linker.

The assembler, defined in the (system vm assembler) module, has a relatively straight-
forward imperative interface. It provides a make-assembler function to instantiate an
assembler and a set of emit-inst procedures to emit instructions of each kind.

The emit-inst procedures are actually generated at compile-time from a machine-
readable description of the VM. With a few exceptions for certain operand types, each
operand of an emit procedure corresponds to an operand of the corresponding instruction.

Consider vector-length, from see Section 9.3.7.9 [Miscellaneous Instructions], page 829.
It is documented as:

```
vector-length u12:dst u12:src

```

Therefore the emit procedure has the form:

```
emit-vector-length asm dst src
```

All emit procedure take the assembler as their first argument, and return no useful
values.

The argument types depend on the operand types. See Section 9.3.7 [Instruction Set],
page 819. Most are integers within a restricted range, though labels are generally expressed
as opaque symbols.

There are a few macro-instructions as well.

```
emit-label asm label
emit-source asm source
emit-cache-current-module! asm module scope
emit-cached-toplevel-box asm dst scope sym bound?
emit-cached-module-box asm dst module-name sym public? bound?
emit-load-constant asm dst constant
```

Macro-instructions to implement caching of top-level variables. The first takes the
current module, in the slot module, and associates it with a cache location identified
by scope. The second takes a scope, and resolves the variable. See Section 9.3.7.2
[Top-Level Environment Instructions], page 821. The last does not need a cached
module, rather taking the module name directly.

Load the Scheme datum constant into dst.
emit-begin-program asm label properties

Delimit the bounds of a procedure, with the given label and the metadata properties.

emit-end-program asm

emit-load-static-procedure asm dst label

Load a procedure with the given label into local dst. This macro-instruction should only be used with procedures without free variables – procedures that are not closures.

emit-begin-standard-arity asm req nlocals alternate

emit-begin-opt-arity asm req opt rest nlocals alternate

emit-begin-kw-arity asm req opt rest kw-indices allow-other-keys? nlocals alternate

emit-end-arity asm

Delimit a clause of a procedure.

emit-br-if-symbol asm slot invert? label

emit-br-if-variable asm slot invert? label

emit-br-if-vector asm slot invert? label

emit-br-if-string asm slot invert? label

emit-br-if-bytevector asm slot invert? label

emit-br-if-bitvector asm slot invert? label

TC7-specific test-and-branch instructions. The TC7 is a 7-bit code that is part of a heap object’s type. See Section 9.2.5 [The SCM Type in Guile], page 808. Also, See Section 9.3.7.6 [Branch Instructions], page 826.

The linker is a complicated beast. Hackers interested in how it works would do well do read Ian Lance Taylor’s series of articles on linkers. Searching the internet should find them easily. From the user’s perspective, there is only one knob to control: whether the resulting image will be written out to a file or not. If the user passes #:to-file? #t as part of the compiler options (see Section 9.4.2 [The Scheme Compiler], page 838), the linker will align the resulting segments on page boundaries, and otherwise not.

link-assembly asm #:page-aligned?==#t

Link an ELF image, and return the bytevector. If page-aligned? is true, Guile will align the segments with different permissions on page-sized boundaries, in order to maximize code sharing between different processes. Otherwise, padding is minimized, to minimize address space consumption.

To write an image to disk, just use put-bytevector from (ice-9 binary-ports).

Compiling object code to the fake language, value, is performed via loading objcode into a program, then executing that thunk with respect to the compilation environment. Normally the environment propagates through the compiler transparently, but users may specify the compilation environment manually as well, as a module. Procedures to load images can be found in the (system vm loader) module:

(use-modules (system vm loader))

load-thunk-from-file file

scm_load_thunk_from_file (file)

Load object code from a file named file. The file will be mapped into memory via mmap, so this is a very fast operation.
load-thunk-from-memory \( bv \)  

[C Function]

scm_load_thunk_from_memory \((bv)\)

Load object code from a bytevector. The data will be copied out of the bytevector in order to ensure proper alignment of embedded Scheme values.

Additionally there are procedures to find the ELF image for a given pointer, or to list all mapped ELF images:

find-mapped-elf-image \( ptr \)  

[Scheme Variable]

Given the integer value \( ptr \), find and return the ELF image that contains that pointer, as a bytevector. If no image is found, return \#f. This routine is mostly used by debuggers and other introspective tools.

all-mapped-elf-images  

[Scheme Variable]

Return all mapped ELF images, as a list of bytevectors.

9.4.6 Writing New High-Level Languages

In order to integrate a new language \( lang \) into Guile’s compiler system, one has to create the module \((language lang spec)\) containing the language definition and referencing the parser, compiler and other routines processing it. The module hierarchy in \((language brainfuck)\) defines a very basic Brainfuck implementation meant to serve as easy-to-understand example on how to do this. See for instance http://en.wikipedia.org/wiki/Brainfuck for more information about the Brainfuck language itself.

9.4.7 Extending the Compiler

At this point we take a detour from the impersonal tone of the rest of the manual. Admit it: if you’ve read this far into the compiler internals manual, you are a junkie. Perhaps a course at your university left you unsated, or perhaps you’ve always harbored a desire to hack the holy of computer science holies: a compiler. Well you’re in good company, and in a good position. Guile’s compiler needs your help.

There are many possible avenues for improving Guile’s compiler. Probably the most important improvement, speed-wise, will be some form of native compilation, both just-in-time and ahead-of-time. This could be done in many ways. Probably the easiest strategy would be to extend the compiled procedure structure to include a pointer to a native code vector, and compile from bytecode to native code at run-time after a procedure is called a certain number of times.

The name of the game is a profiling-based harvest of the low-hanging fruit, running programs of interest under a system-level profiler and determining which improvements would give the most bang for the buck. It’s really getting to the point though that native compilation is the next step.

The compiler also needs help at the top end, enhancing the Scheme that it knows to also understand R6RS, and adding new high-level compilers. We have JavaScript and Emacs Lisp mostly complete, but they could use some love; Lua would be nice as well, but whatever language it is that strikes your fancy would be welcome too.

Compilers are for hacking, not for admiring or for complaining about. Get to it!
Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released
under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING
You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,
be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their
titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.
7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.
10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.
ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the
document and put the following copyright and license notices just after the title page:

Copyright (C) year your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts. A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the
‘‘with...Texts.’’ line with this:

with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.

If you have Invariant Sections without Cover Texts, or some other combination of the
three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing
these examples in parallel under your choice of free software license, such as the GNU
General Public License, to permit their use in free software.
Concept Index

This index contains concepts, keywords and non-Scheme names for several features, to make it easier to locate the desired sections.

! .......................... 381
!
    # ! .......................... 381
    # .() .......................... 601

( ice-9 match) ................. 691

. .......................... 48
    .guile ......................... 37
    .guile file, not loading ........ 696
    .guile_history ................ 696
    .inputrc ........................ 696

/                        /etc/hosts .................................. 529
                        /etc/protocols .......................... 531
                        /etc/services .......................... 532

A .......................... absolute file name .................. 506
    addinfo object type ............ 527
    affinity, CPU .......................... 520
    alist .................................. 229, 588
    argument specialize ............... 615
    arguments (command line) ........ 35
    arity, variable .................... 255, 603
    array cell .......................... 208
    array frame .......................... 208
    array slice .......................... 208
    association List .................... 229
    association list .................... 588
    asynchronous interrupts .......... 439
    asyncs ............................. 439
    atomic time .......................... 608
    autoload ............................ 409
    automatic compilation ............. 386
    automatically-managed memory ... 402

B .......................... begin .............................. 298
    binary input ......................... 330
    binary output ....................... 331
    binary port .......................... 670
    binding renamer ..................... 408
    bindir ................................ 451
    bitwise logical ..................... 642
    block comments ..................... 381
    BOM .................................. 354
    Breakpoints .......................... 480, 488
    Buffered input ...................... 720
    buildstamp .......................... 451
    byte order ........................... 193
    byte order mark ..................... 354
    bytevector ........................... 193

C .......................... callbacks .......................... 435
    canonical host type .................. 452
    case .................................. 299
    case folding .......................... 131
    chaining environments ................. 27
    character encoding ........................ 159
    charset ............................. 139
    child processes ..................... 514
    class .............................. 759
    Closing ports ...................... 329
    closure ............................. 26
    code coverage ....................... 489
    Code coverage ........................ 480
    code point ........................... 129
    code point, designated .............. 129
    code point, reserved ................. 129
    codec .................................. 670
    codeset ............................. 139
    command line ........................ 512, 513
    command line history ................ 695
    command-line arguments .............. 35
    Command-line Options ............... 35
    commands ............................. 49
    composable continuations .......... 303
    cond .................................. 299
    condition variable .................. 442
    conditional evaluation .............. 299
    conditions ........................... 620
    conservative garbage collection ..... 402
    continuation, CPS .................... 845
    continuation, escape ................ 305
    continuations ....................... 307
conversion strategy, port .................................. 332
Cooperative REPL server .................................. 400
copying ....................................................... 1
coverage ..................................................... 489
CPS .......................................................... 843
CPS, first-order ............................................ 847
CPS, higher-order .......................................... 847
cultural conventions ...................................... 459
curly-infix ................................................... 647
curly-infix-and-bracket-lists ............................ 647
current directory ........................................... 514
custom binary input ports .................................. 345
custom binary input/output ports ......................... 346
custom binary output ports ................................ 346

database ...................................................... 229
datadir ........................................................ 451
date .......................................................... 608, 611
date conversion ............................................. 612
date to string ............................................... 613
date, from string .......................................... 614
debug options .............................................. 480
Debugging ................................................... 468
debugging virtual machine (command line) ............ 37
decoding error .............................................. 332
Default ports .............................................. 340
default slot value ......................................... 761
definition splicing ......................................... 298
delayed evaluation ........................................ 393
delimited continuations .................................. 303
designated code point .................................... 129
device file .................................................... 505
directory contents ....................................... 504
directory traversal ....................................... 711
domain-specific language ................................. 261
dominate, CPS ............................................... 846
DSL .......................................................... 261
duplicate binding .......................................... 410, 411
duplicate binding handlers ................................ 411

default ports .............................................. 340
default slot value ......................................... 761
definition splicing ......................................... 298
delayed evaluation ........................................ 393
delimited continuations .................................. 303
designated code point .................................... 129
device file .................................................... 505
directory contents ....................................... 504
directory traversal ....................................... 711
domain-specific language ................................. 261
dominate, CPS ............................................... 846
DSL .......................................................... 261
duplicate binding .......................................... 410, 411
duplicate binding handlers ................................ 411

effectiveness ................................................. 59
Emacs .......................................................... 56
emacs regexp ............................................... 355
embedded domain-specific language ..................... 261
encapsulation ............................................... 406
enclosed array ............................................. 208
encoding ..................................................... 392
encoding error ............................................. 332
encryption ................................................... 543
End of file object ......................................... 330
end-of-line style ......................................... 670
endianness .................................................. 193
environment ................................................ 26, 421, 513
evironment variables ..................................... 38, 647
environment, local ........................................ 27
evironment, top level ...................................... 26
equality ...................................................... 283
errno .......................................................... 492, 493
error handling ............................................. 311
error-signal .................................................. 326
evaluate expression, command-line argument .......... 35
evaluation, top level ...................................... 26
exception handling ....................................... 311
exceptions .................................................. 620
eexec_prefix .................................................. 451
export ........................................................ 410
expression sequencing .................................. 298
expression, CPS ........................................... 845
extensiondir ............................................... 60, 426

F
fdes finalizers ............................................... 500
fi .............................................................. 422
file descriptor ............................................. 493
file descriptor finalizers ................................ 500
file locking .................................................. 499
file name separator ....................................... 506
File port ..................................................... 341
file system ................................................ 500
file system combinator .................................. 712
file system traversal ..................................... 711
file tree walk ............................................. 711
finalization ................................................ 78, 244, 246
finalizer ..................................................... 78, 244, 246
finalizers, file descriptor ............................... 500
fine-grain parallelism .................................... 447
first-order CPS ............................................ 847
fluids .......................................................... 321
fold-case ..................................................... 381
foreign function interface ................................ 422
foreign object ............................................. 243
formatted output ......................................... 700
frame rank .................................................. 208
functional setters ........................................ 219
futures ....................................................... 447
G
GC-managed memory ...................................... 402
GDB support ................................................ 489
Geiser ......................................................... 56
general cond clause ...................................... 300
GNU triplet .................................................. 452
GPL ............................................................ 1
group file .................................................... 507
guardians, testing for GC’d objects ..................... 293
guild .......................................................... 56
Guile threads ............................................... 436
guile-2 SRFI-0 feature ................................... 578
guile-snarf deprecated macros ......................... 102
guile-snarf example ....................................... 82
S
sameness ........................................ 283
sbindir ........................................ 451
Scheme Shell ...................................... 728
SCM data type .................................. 100
script mode ..................................... 35
SCSH ............................................ 728
search and replace .............................. 359
sequencing ....................................... 298
service name lookup ............................ 527
services .......................................... 532
setter ............................................. 259
Setting breakpoints ................................ 488
Setting breakpoints ............................. 167
shadowsing an imported variable binding ...... 7
shardestatedir ................................... 451
shell .............................................. 38
signal ............................................ 520
site ............................................... 57
site path ......................................... 57
sitepath .......................................... 59
SLIB .............................................. 491
slot ............................................... 750
smob ............................................. 245
socket ............................................ 535
socket address .................................. 533
socket client example ........................... 540
socket examples .................................. 540
socket server example ........................... 541
Soft port ......................................... 347
sorting ............................................ 286
sorting lists ...................................... 286
sorting vectors .................................... 286
source file encoding ............................. 392
source properties ................................ 471
specialize parameter ............................ 615
splicing .......................................... 298
srchdir ........................................... 451
SRFI .............................................. 576
SRFI-0 ............................................ 577
SRFI-1 ............................................ 578
SRFI-10 ......................................... 601
SRFI-105 ......................................... 647
SRFI-11 .......................................... 603
SRFI-111 ......................................... 648
SRFI-13 .......................................... 603
SRFI-14 .......................................... 603
SRFI-16 .......................................... 255, 603
SRFI-17 .......................................... 603
SRFI-18 .......................................... 604
SRFI-19 .......................................... 608
SRFI-2 ............................................ 591
SRFI-23 .......................................... 615
SRFI-26 .......................................... 615
SRFI-27 .......................................... 617, 618
SRFI-28 .......................................... 619
SRFI-30 .......................................... 619
SRFI-30 block comments ........................ 381
SRFI-31 ........................................... 619
SRFI-34 ........................................... 619
SRFI-35 ........................................... 620
SRFI-37 ........................................... 622
SRFI-38 ........................................... 623
SRFI-39 ........................................... 324, 325, 624
SRFI-4 ............................................. 592
SRFI-41 .......................................... 625
SRFI-42 .......................................... 634
SRFI-43 .......................................... 634
SRFI-45 .......................................... 639
SRFI-46 .......................................... 641
SRFI-55 .......................................... 641
SRFI-6 ............................................ 601
SRFI-60 .......................................... 642
SRFI-61 .......................................... 300
SRFI-62 .......................................... 643
SRFI-64 .......................................... 643
SRFI-67 .......................................... 643
SRFI-69 .......................................... 644
SRFI-71 .......................................... 646
SRFI-8 ............................................ 601
SRFI-87 .......................................... 646
SRFI-88 .......................................... 646
SRFI-88 keyword syntax ........................ 175
SRFI-9 ............................................ 218
SRFI-98 .......................................... 647
stack overflow .................................... 477
stack-overflow .................................... 326
standard error output ............................ 340
standard input .................................... 340
standard output ................................... 340
startup (command-line arguments) ............ 35
streams .......................................... 718
String port ....................................... 344
string to date .................................... 614
string, from date .................................. 613
structure ......................................... 217
switches (command line) ....................... 35
SXML pattern matching .......................... 723
symbolic math ..................................... 492
sysconfdir ....................................... 451
system clock ..................................... 609
system name ..................................... 541
system-error ..................................... 326
T
tail calls ......................................... 24
TAI ............................................... 608, 609
temporary file .................................... 505, 506
term, CPS .......................................... 845
terminal .......................................... 523, 524
textual input ..................................... 333
textual output .................................... 333
textual port ....................................... 670
thread time ....................................... 609
threads .......................................... 436
time ........................................... 509, 608, 609
time conversion ............................... 612
time formatting ................................ 511
time parsing .................................... 511
top level environment ............................ 26
top_srcdir ......................................... 451
Trace .................................................. 480
Tracepoints .......................................... 488
Tracing ............................................... 480
transcoder ............................................. 670
transformation ...................................... 261
transient intmaps .................................. 851
Traps ................................................... 480
truncated printing .................................. 699
Types of ports ....................................... 341

U
Unicde code point .................. 129
Unicde string encoding ............ 199
universal time ............................. 608
unless ............................................. 299
user information ............................ 507
UTC .................................................. 608, 609

V
variable arity ................................. 255, 603
variable definition ......................... 293
variable, local ............................... 27
vcell .............................................. 26
VHash ............................................. 235
vlist ............................................. 215
VList-based hash lists ...................... 235
VM hooks ........................................ 480
VM trace level ................................. 482
Void port ......................................... 347

W
warnings, compiler .......................... 387
Web ............................................. 543
when .............................................. 299
wizards .......................................... 56
word order ....................................... 193
wrapped pointer types .................... 431
wrong-number-of-args ...................... 326
wrong-type-arg .................................. 326
WWW ................................................ 543
Procedure Index

This is an alphabetical list of all the procedures and macros in Guile. It also includes Guile’s Autoconf macros.

When looking for a particular procedure, please look under its Scheme name as well as under its C name. The C name can be constructed from the Scheme names by a simple transformation described in the section See Section 6.1 [API Overview], page 99.

#
#:accessor 763
#:allocation 763
#:class 764
#:each-subclass 764
#:getter 763
#:init-form 761
#:init-keyword 761
#:init-thunk 761
#:init-value 761
#:instance 763
#:metaclass 760
#:name 760
#:setter 763
#:slot-ref 764, 765
#:slot-set! 764, 765
#:virtual 764

%
%char-set-dump 306
%default-port-conversion-strategy 332
%library-dir 451
%make-void-port 347
%package-data-dir 451
%read-delimited! 339
%read-line 339
%search-load-path 390
%site-ccache-dir 57, 451
%site-dir 57, 451
%string-dump 163

&
&action 667
&condition 666
&error 667
&o... 668
&o-decoding 671
&o-encoding 671
&o-file-already-exists 669
&o-file-encodes 669
&o-file-encodes-... 669
&o-file-does-not-exist 669
&o-file-is-read-only 669
&o-file-protection 669
&o-filename 669
&o-invalid-position 669
&o-port 669
&o-read 668
&o-write 669
&implementation-restriction 668
&irritants 667
&lexical 668
&message 667
&no-infinities 684
&no-nans 684
&non-continuable 668
&non-continuable 684
&serious 667
&syntax 668
&undefined 668
&violation 667
&warning 667
&who 667

,' 379

(oop goops) 759

* 119, 655
*scm_to_latin1_string 162
*scm_to_string 162
*scm_to_utf32_string 162
*scm_to_utf8_string 162

+ 119, 655

, 380
,0 380

- 119, 655
->char-set 137
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>char-locale-downcase</td>
<td>461</td>
</tr>
<tr>
<td>char-locale-titlecase</td>
<td>461</td>
</tr>
<tr>
<td>char-locale-upcase</td>
<td>461</td>
</tr>
<tr>
<td>char-locale?</td>
<td>460</td>
</tr>
<tr>
<td>char-locale?</td>
<td>460</td>
</tr>
<tr>
<td>char-lower-case?</td>
<td>132, 658</td>
</tr>
<tr>
<td>char-numeric?</td>
<td>132, 658</td>
</tr>
<tr>
<td>char-ready?</td>
<td>348</td>
</tr>
<tr>
<td>char-set</td>
<td>135</td>
</tr>
<tr>
<td>char-set-&gt;list</td>
<td>137</td>
</tr>
<tr>
<td>char-set-&gt;string</td>
<td>137</td>
</tr>
<tr>
<td>char-set-adjoin</td>
<td>138</td>
</tr>
<tr>
<td>char-set-adjoin!</td>
<td>138</td>
</tr>
<tr>
<td>char-set-complement</td>
<td>138</td>
</tr>
<tr>
<td>char-set-complement!</td>
<td>139</td>
</tr>
<tr>
<td>char-set-contains?</td>
<td>137</td>
</tr>
<tr>
<td>char-set-copy</td>
<td>135</td>
</tr>
<tr>
<td>char-set-count</td>
<td>137</td>
</tr>
<tr>
<td>char-set-cursor</td>
<td>134</td>
</tr>
<tr>
<td>char-set-cursor-next</td>
<td>134</td>
</tr>
<tr>
<td>char-set-delete</td>
<td>138</td>
</tr>
<tr>
<td>char-set-delete!</td>
<td>138</td>
</tr>
<tr>
<td>char-set-diff-intersection</td>
<td>139</td>
</tr>
<tr>
<td>char-set-diff-intersection!</td>
<td>139</td>
</tr>
<tr>
<td>char-set-difference</td>
<td>138</td>
</tr>
<tr>
<td>char-set-difference!</td>
<td>139</td>
</tr>
<tr>
<td>char-set-every</td>
<td>137</td>
</tr>
<tr>
<td>char-set-filter</td>
<td>136</td>
</tr>
<tr>
<td>char-set-filter!</td>
<td>136</td>
</tr>
<tr>
<td>char-set-fold</td>
<td>135</td>
</tr>
<tr>
<td>char-set-for-each</td>
<td>135</td>
</tr>
<tr>
<td>char-set-hash</td>
<td>134</td>
</tr>
<tr>
<td>char-set-intersection</td>
<td>138</td>
</tr>
<tr>
<td>char-set-intersection!</td>
<td>139</td>
</tr>
<tr>
<td>char-set-map</td>
<td>135</td>
</tr>
<tr>
<td>char-set-ref</td>
<td>134</td>
</tr>
<tr>
<td>char-set-size</td>
<td>137</td>
</tr>
<tr>
<td>char-set-unfold</td>
<td>135</td>
</tr>
<tr>
<td>char-set-unfold!</td>
<td>135</td>
</tr>
<tr>
<td>char-set-union</td>
<td>138</td>
</tr>
<tr>
<td>char-set-union!</td>
<td>139</td>
</tr>
<tr>
<td>char-setxor</td>
<td>139</td>
</tr>
<tr>
<td>char-setxor!</td>
<td>139</td>
</tr>
<tr>
<td>char-set&lt;=</td>
<td>134</td>
</tr>
<tr>
<td>char-set=</td>
<td>134</td>
</tr>
<tr>
<td>char-set?</td>
<td>134</td>
</tr>
<tr>
<td>char-title-case?</td>
<td>658</td>
</tr>
<tr>
<td>char-titlecase</td>
<td>133, 657</td>
</tr>
<tr>
<td>char-upcase</td>
<td>133, 657</td>
</tr>
<tr>
<td>char-upper-case?</td>
<td>132, 658</td>
</tr>
<tr>
<td>char-whitespace?</td>
<td>132, 658</td>
</tr>
<tr>
<td>char=&lt;?</td>
<td>131, 651</td>
</tr>
<tr>
<td>char=?</td>
<td>131, 651</td>
</tr>
<tr>
<td>char&gt;=?</td>
<td>131, 651</td>
</tr>
<tr>
<td>char=?</td>
<td>131, 651</td>
</tr>
<tr>
<td>char?</td>
<td>131, 650</td>
</tr>
<tr>
<td>chdir</td>
<td>514</td>
</tr>
<tr>
<td>chmod</td>
<td>503</td>
</tr>
<tr>
<td>chown</td>
<td>503</td>
</tr>
<tr>
<td>chroot</td>
<td>514</td>
</tr>
<tr>
<td>circular-list</td>
<td>579</td>
</tr>
<tr>
<td>circular-list?</td>
<td>579</td>
</tr>
<tr>
<td>class</td>
<td>788</td>
</tr>
<tr>
<td>class-direct-methods</td>
<td>776</td>
</tr>
<tr>
<td>class-direct-slots</td>
<td>776</td>
</tr>
<tr>
<td>class-direct-subclasses</td>
<td>776</td>
</tr>
<tr>
<td>class-direct-super</td>
<td>776</td>
</tr>
<tr>
<td>class-methods</td>
<td>777</td>
</tr>
<tr>
<td>class-name</td>
<td>776</td>
</tr>
<tr>
<td>class-of</td>
<td>777, 831</td>
</tr>
<tr>
<td>class-precedence-list</td>
<td>777</td>
</tr>
<tr>
<td>class-redefinition</td>
<td>796</td>
</tr>
<tr>
<td>class-slot-definition</td>
<td>777</td>
</tr>
<tr>
<td>class-slot-set!</td>
<td>781</td>
</tr>
<tr>
<td>class-slot-ref</td>
<td>781</td>
</tr>
<tr>
<td>close</td>
<td>496</td>
</tr>
<tr>
<td>close-fdes</td>
<td>496</td>
</tr>
<tr>
<td>close-input-port</td>
<td>678</td>
</tr>
<tr>
<td>close-output-port</td>
<td>678</td>
</tr>
<tr>
<td>close-pipe</td>
<td>525</td>
</tr>
<tr>
<td>close-port</td>
<td>329</td>
</tr>
<tr>
<td>close-server</td>
<td>567</td>
</tr>
<tr>
<td>closedir</td>
<td>505</td>
</tr>
<tr>
<td>collapse-repeated-chars</td>
<td>753</td>
</tr>
<tr>
<td>command-line</td>
<td>512, 679</td>
</tr>
<tr>
<td>compile</td>
<td>50, 387</td>
</tr>
<tr>
<td>compile-file</td>
<td>50, 388</td>
</tr>
<tr>
<td>compile-peg-pattern</td>
<td>367</td>
</tr>
<tr>
<td>compiled-file-name</td>
<td>388</td>
</tr>
<tr>
<td>complex</td>
<td>113, 653</td>
</tr>
<tr>
<td>compose</td>
<td>258</td>
</tr>
<tr>
<td>compose-continuation</td>
<td>825</td>
</tr>
<tr>
<td>compute-std-cpl</td>
<td>791</td>
</tr>
<tr>
<td>concatenate</td>
<td>581</td>
</tr>
<tr>
<td>concatenate!</td>
<td>581</td>
</tr>
<tr>
<td>cond</td>
<td>300, 652</td>
</tr>
<tr>
<td>cond-expand</td>
<td>577</td>
</tr>
<tr>
<td>condition</td>
<td>621, 666</td>
</tr>
<tr>
<td>condition-accessor</td>
<td>666</td>
</tr>
<tr>
<td>condition-has-type?</td>
<td>620</td>
</tr>
<tr>
<td>condition-irritants</td>
<td>667</td>
</tr>
<tr>
<td>condition-message</td>
<td>621, 667</td>
</tr>
<tr>
<td>condition-predicate</td>
<td>666</td>
</tr>
<tr>
<td>condition-ref</td>
<td>620</td>
</tr>
<tr>
<td>condition-type?</td>
<td>620</td>
</tr>
<tr>
<td>condition-variable-broadcast!</td>
<td>607</td>
</tr>
<tr>
<td>condition-variable-name</td>
<td>607</td>
</tr>
<tr>
<td>condition-variable-signal!</td>
<td>607</td>
</tr>
<tr>
<td>condition-variable-specific</td>
<td>607</td>
</tr>
<tr>
<td>condition-variable-specific-set!</td>
<td>607</td>
</tr>
<tr>
<td>condition-variable?</td>
<td>445, 606</td>
</tr>
</tbody>
</table>
condition-who                      668
collection                     666
code                           537
cons                            179, 651, 831
cons*                           182, 660
cons-source                    472
const                           257
context-flatten                 369
continuation-call               825
continue                        302
copy-bit                        642
copy-bit-field                  642
copy-file                       503
copy-random-state               127
copy-time                       610
copy-tree                       288
cos                             124, 653
cosh                            124
count                           582
coverage-data->lcov             489
coverage-data?                  489
crypt                           543
ctermid                         524
current-date                    612
current-dynamic-state           323
current-error-port              340, 674
current-exception-handler       607
current-filename                472
current-http-proxy              565
current-input-port              340, 674
current-julian-day              612
current-load-port               389
current-modified-julian-day      612
current-module                  417, 821
current-output-port             340, 674
current-processor-count         438
current-read-waiter             354
current-source-location         472
current-ssax-error-port         738
current-thread                  436, 604, 829
current-time                    509, 607, 610
current-write-waiter            354
Cut                             616
cute                            616
date->second                    611
date-week-day                   611
date-week-number                612
date-year                       611
date-year-day                   611
date-zone-offset                611
date?                           611
datum->random-state             128
datum->random-state             270, 686
date->time-utc                  611
date->time-tai                  611
date->julian-day                611
date->time-utc                  184, 587
date->time-tai                  185, 587
date?                           611
datum->random-state             128
datum->random-state             270, 686
date->time-utc                  611
date->time-tai                  611
date->julian-day                611
date->time-utc                  184, 587
date->time-tai                  185, 587

date->julian-day                612
date->modified-julian-day        612
date->string                    613
date->time-monotonic            612
date->time-tai                  612
date->time-utc                  612
date-day                        611
date-hour                       611
date-minute                     611
date-month                      611
date-nanosecond                 611
date->second                    611
date-week-day                   611
date-week-number                612
date-year                       611
date-year-day                   611
date-zone-offset                611
date?                           611
datum->random-state             128
datum->random-state             270, 686
date->time-utc                  611
date->time-tai                  611
date->julian-day                611
date->time-utc                  184, 587
date->time-tai                  185, 587

date->julian-day                612
date->modified-julian-day        612
date->string                    613
date->time-monotonic            612
date->time-tai                  612
date->time-utc                  612
date-day                        611
date-hour                       611
date-minute                     611
date-month                      611
date-nanosecond                 611

date->second                    611
date-week-day                   611
date-week-number                612
date-year                       611
date-year-day                   611
date-zone-offset                611
date?                           611
datum->random-state             128
datum->random-state             270, 686
date->time-utc                  611
date->time-tai                  611
date->julian-day                611
date->time-utc                  184, 587
date->time-tai                  185, 587
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete-duplicates</td>
<td>588</td>
</tr>
<tr>
<td>delete-duplicates!</td>
<td>588</td>
</tr>
<tr>
<td>delete-file</td>
<td>503</td>
</tr>
<tr>
<td>delete-trap!</td>
<td>487</td>
</tr>
<tr>
<td>delete!</td>
<td>185</td>
</tr>
<tr>
<td>delq</td>
<td>184</td>
</tr>
<tr>
<td>delq!</td>
<td>185</td>
</tr>
<tr>
<td>delv</td>
<td>184</td>
</tr>
<tr>
<td>delv!</td>
<td>185</td>
</tr>
<tr>
<td>denominator</td>
<td>112, 653</td>
</tr>
<tr>
<td>deq!</td>
<td>717</td>
</tr>
<tr>
<td>dereference-pointer</td>
<td>430</td>
</tr>
<tr>
<td>describe</td>
<td>50</td>
</tr>
<tr>
<td>directory-stream?</td>
<td>505</td>
</tr>
<tr>
<td>dirname</td>
<td>506</td>
</tr>
<tr>
<td>disable-trap!</td>
<td>487</td>
</tr>
<tr>
<td>disable-value-history!</td>
<td>40</td>
</tr>
<tr>
<td>disassemble</td>
<td>51</td>
</tr>
<tr>
<td>disassemble-file</td>
<td>51</td>
</tr>
<tr>
<td>display</td>
<td>383, 678, 783</td>
</tr>
<tr>
<td>display-application</td>
<td>471</td>
</tr>
<tr>
<td>display-backtrace</td>
<td>469</td>
</tr>
<tr>
<td>display-error</td>
<td>326</td>
</tr>
<tr>
<td>div</td>
<td>655, 833</td>
</tr>
<tr>
<td>div-and-mod</td>
<td>655</td>
</tr>
<tr>
<td>div0</td>
<td>655</td>
</tr>
<tr>
<td>div0-and-mod0</td>
<td>655</td>
</tr>
<tr>
<td>do</td>
<td>301, 660</td>
</tr>
<tr>
<td>dotted-list?</td>
<td>580</td>
</tr>
<tr>
<td>doubly-weak-hash-table?</td>
<td>405</td>
</tr>
<tr>
<td>down</td>
<td>51</td>
</tr>
<tr>
<td>drain-input</td>
<td>337</td>
</tr>
<tr>
<td>drop</td>
<td>581, 830</td>
</tr>
<tr>
<td>drop-right</td>
<td>581</td>
</tr>
<tr>
<td>drop-right!</td>
<td>581</td>
</tr>
<tr>
<td>drop-while</td>
<td>586</td>
</tr>
<tr>
<td>dup</td>
<td>497</td>
</tr>
<tr>
<td>dup-&gt;+ides</td>
<td>497</td>
</tr>
<tr>
<td>dup-&gt;inport</td>
<td>497</td>
</tr>
<tr>
<td>dup-&gt;outport</td>
<td>497</td>
</tr>
<tr>
<td>dup-&gt;port</td>
<td>497</td>
</tr>
<tr>
<td>dup2</td>
<td>497</td>
</tr>
<tr>
<td>duplicate-port</td>
<td>497</td>
</tr>
<tr>
<td>dynamic-call</td>
<td>424</td>
</tr>
<tr>
<td>dynamic-func</td>
<td>423</td>
</tr>
<tr>
<td>dynamic-link</td>
<td>422</td>
</tr>
<tr>
<td>dynamic-object?</td>
<td>423</td>
</tr>
<tr>
<td>dynamic-pointer</td>
<td>429</td>
</tr>
<tr>
<td>dynamic-state?</td>
<td>323</td>
</tr>
<tr>
<td>dynamic-unlink</td>
<td>423</td>
</tr>
<tr>
<td>dynamic-wind</td>
<td>318, 657</td>
</tr>
</tbody>
</table>

**E**

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>eager</td>
<td>640</td>
</tr>
<tr>
<td>effective-version</td>
<td>450</td>
</tr>
<tr>
<td>eighth</td>
<td>580</td>
</tr>
<tr>
<td>emit-begin-kw-arity</td>
<td>854</td>
</tr>
<tr>
<td>emit-begin-opt-arity</td>
<td>854</td>
</tr>
<tr>
<td>emit-begin-program</td>
<td>854</td>
</tr>
<tr>
<td>emit-begin-standard-arity</td>
<td>854</td>
</tr>
<tr>
<td>emit-br-if-bitvector</td>
<td>854</td>
</tr>
<tr>
<td>emit-br-if-bytevector</td>
<td>854</td>
</tr>
<tr>
<td>emit-br-if-string</td>
<td>854</td>
</tr>
<tr>
<td>emit-br-if-symbol</td>
<td>854</td>
</tr>
<tr>
<td>emit-br-if-variable</td>
<td>854</td>
</tr>
<tr>
<td>emit-br-if-vector</td>
<td>854</td>
</tr>
<tr>
<td>emit-cache-current-module!</td>
<td>853</td>
</tr>
<tr>
<td>emit-cached-module-box</td>
<td>853</td>
</tr>
<tr>
<td>emit-cached-toplevel-box</td>
<td>853</td>
</tr>
<tr>
<td>emit-end-arity</td>
<td>854</td>
</tr>
<tr>
<td>emit-end-program</td>
<td>854</td>
</tr>
<tr>
<td>emit-label</td>
<td>853</td>
</tr>
<tr>
<td>emit-load-constant</td>
<td>853</td>
</tr>
<tr>
<td>emit-load-static-procedure</td>
<td>854</td>
</tr>
<tr>
<td>emit-source</td>
<td>853</td>
</tr>
<tr>
<td>emit-vector-length</td>
<td>853</td>
</tr>
<tr>
<td>enable-primitive-generic!</td>
<td>768</td>
</tr>
<tr>
<td>enable-trap!</td>
<td>487</td>
</tr>
<tr>
<td>enable-value-history!</td>
<td>49</td>
</tr>
<tr>
<td>encode-and-join-uri-path</td>
<td>547</td>
</tr>
<tr>
<td>end-of-char-set?</td>
<td>135</td>
</tr>
<tr>
<td>endgrent</td>
<td>509</td>
</tr>
<tr>
<td>endhostent</td>
<td>530</td>
</tr>
<tr>
<td>endianness</td>
<td>194</td>
</tr>
<tr>
<td>endnetent</td>
<td>531</td>
</tr>
<tr>
<td>endprotoent</td>
<td>532</td>
</tr>
<tr>
<td>endpwent</td>
<td>508</td>
</tr>
<tr>
<td>endservent</td>
<td>533</td>
</tr>
<tr>
<td>enq!</td>
<td>717</td>
</tr>
<tr>
<td>ensure-accessor</td>
<td>794</td>
</tr>
<tr>
<td>ensure-generic</td>
<td>793</td>
</tr>
<tr>
<td>ensure-metaclass</td>
<td>789</td>
</tr>
<tr>
<td>enum-set-&gt;list</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-complement</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-constructor</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-difference</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-indexer</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-intersection</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-member?</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-projection</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-subset?</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-union</td>
<td>689</td>
</tr>
<tr>
<td>enum-set-universe</td>
<td>689</td>
</tr>
<tr>
<td>enum-set=?</td>
<td>689</td>
</tr>
<tr>
<td>environ</td>
<td>514</td>
</tr>
<tr>
<td>environment</td>
<td>690</td>
</tr>
<tr>
<td>eof-object</td>
<td>673, 678</td>
</tr>
<tr>
<td>eof-object?</td>
<td>330, 673, 678</td>
</tr>
<tr>
<td>eol-style</td>
<td>670</td>
</tr>
<tr>
<td>eq?</td>
<td>283, 653</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------</td>
<td>------</td>
</tr>
<tr>
<td>fdes-&gt;inport</td>
<td>495</td>
</tr>
<tr>
<td>fdes-&gt;outport</td>
<td>495</td>
</tr>
<tr>
<td>fdes-&gt;posts</td>
<td>495</td>
</tr>
<tr>
<td>fdv</td>
<td>836</td>
</tr>
<tr>
<td>fdopen</td>
<td>494</td>
</tr>
<tr>
<td>feature?</td>
<td>452</td>
</tr>
<tr>
<td>file-encoding</td>
<td>393</td>
</tr>
<tr>
<td>file-exists?</td>
<td>506</td>
</tr>
<tr>
<td>file-name-separator?</td>
<td>507</td>
</tr>
<tr>
<td>file-options</td>
<td>676</td>
</tr>
<tr>
<td>file-port?</td>
<td>344</td>
</tr>
<tr>
<td>file-system-fold</td>
<td>713</td>
</tr>
<tr>
<td>file-system-tree</td>
<td>711</td>
</tr>
<tr>
<td>filename-completion-function</td>
<td>698</td>
</tr>
<tr>
<td>fileno</td>
<td>494</td>
</tr>
<tr>
<td>fill-string</td>
<td>754</td>
</tr>
<tr>
<td>filter</td>
<td>185, 659, 744</td>
</tr>
<tr>
<td>filter!</td>
<td>185</td>
</tr>
<tr>
<td>filter-empty-elements</td>
<td>751</td>
</tr>
<tr>
<td>filter-map</td>
<td>585</td>
</tr>
<tr>
<td>find</td>
<td>586, 659</td>
</tr>
<tr>
<td>find-mapped-elf-image</td>
<td>855</td>
</tr>
<tr>
<td>find-string-from-port?</td>
<td>748</td>
</tr>
<tr>
<td>find-tail</td>
<td>586</td>
</tr>
<tr>
<td>finish</td>
<td>52</td>
</tr>
<tr>
<td>infinite?</td>
<td>112, 656</td>
</tr>
<tr>
<td>first</td>
<td>580</td>
</tr>
<tr>
<td>first-set-bit</td>
<td>642</td>
</tr>
<tr>
<td>fixnum-&gt;f64vector</td>
<td>684</td>
</tr>
<tr>
<td>fixnum-width</td>
<td>679</td>
</tr>
<tr>
<td>fixnum?</td>
<td>679</td>
</tr>
<tr>
<td>f1*</td>
<td>683</td>
</tr>
<tr>
<td>f1+</td>
<td>683</td>
</tr>
<tr>
<td>f1-</td>
<td>683</td>
</tr>
<tr>
<td>f1/</td>
<td>683</td>
</tr>
<tr>
<td>f1&lt;=?</td>
<td>682</td>
</tr>
<tr>
<td>f1&lt;?</td>
<td>682</td>
</tr>
<tr>
<td>f1=?</td>
<td>682</td>
</tr>
<tr>
<td>f1&gt;?</td>
<td>682</td>
</tr>
<tr>
<td>f1b</td>
<td>683</td>
</tr>
<tr>
<td>flabs</td>
<td>683</td>
</tr>
<tr>
<td>flacos</td>
<td>683</td>
</tr>
<tr>
<td>flasin</td>
<td>683</td>
</tr>
<tr>
<td>flatan</td>
<td>683</td>
</tr>
<tr>
<td>fceiling</td>
<td>683</td>
</tr>
<tr>
<td>fcos</td>
<td>683</td>
</tr>
<tr>
<td>fdenominator</td>
<td>683</td>
</tr>
<tr>
<td>fdiv</td>
<td>683</td>
</tr>
<tr>
<td>fdiv-and-mod</td>
<td>683</td>
</tr>
<tr>
<td>fdiv0</td>
<td>683</td>
</tr>
<tr>
<td>fdiv0-and-mod0</td>
<td>683</td>
</tr>
<tr>
<td>fdivmod</td>
<td>683</td>
</tr>
<tr>
<td>fleven</td>
<td>682</td>
</tr>
<tr>
<td>flexp</td>
<td>683</td>
</tr>
<tr>
<td>flexpt</td>
<td>684</td>
</tr>
<tr>
<td>fffinite?</td>
<td>682</td>
</tr>
<tr>
<td>fffloor</td>
<td>683</td>
</tr>
</tbody>
</table>

**F**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>f32vector</td>
<td>595</td>
</tr>
<tr>
<td>f32vector-&gt;list</td>
<td>597</td>
</tr>
<tr>
<td>f32vector-length</td>
<td>596</td>
</tr>
<tr>
<td>f32vector-ref</td>
<td>596</td>
</tr>
<tr>
<td>f32vector-set!</td>
<td>597</td>
</tr>
<tr>
<td>f32vector?</td>
<td>594</td>
</tr>
<tr>
<td>f64-&gt;scm</td>
<td>836</td>
</tr>
<tr>
<td>f64vector</td>
<td>595</td>
</tr>
<tr>
<td>f64vector-&gt;list</td>
<td>597</td>
</tr>
<tr>
<td>f64vector-length</td>
<td>596</td>
</tr>
<tr>
<td>f64vector-ref</td>
<td>596</td>
</tr>
<tr>
<td>f64vector-set!</td>
<td>597</td>
</tr>
<tr>
<td>f64vector?</td>
<td>594</td>
</tr>
<tr>
<td>fadd</td>
<td>836</td>
</tr>
<tr>
<td>false-if-exception</td>
<td>317</td>
</tr>
<tr>
<td>fchmod</td>
<td>503</td>
</tr>
<tr>
<td>fchown</td>
<td>503</td>
</tr>
<tr>
<td>fcnt1</td>
<td>498</td>
</tr>
<tr>
<td>fdes-&gt;inport</td>
<td>495</td>
</tr>
</tbody>
</table>
fxxor ............................................. 681
fzero? ......................................... 680

G

gc ............................................... 53, 401
gc-live-object-stats ...................... 401
gc-stats ........................................ 401
gcd ........................................... 116, 654
gcprof .......................................... 733
generate-temporaries ..................... 272, 686
generic-function-name .................... 779
gensym ......................................... 170
get-bytevector-all ....................... 331, 675
get-bytevector-n ............................ 330, 674
get-bytevector-n! ........................... 330, 674
get-bytevector-some ..................... 330, 674
get-bytevector-some! ........................ 330
get-char ...................................... 333, 675
get-datum ..................................... 675
get-environment-variable ................. 647
get-environment-variables ............... 647
get-internal-real-time ................... 512
get-internal-run-time .................... 512
get-line ....................................... 334, 675
get-output-string ......................... 345
get-string-all ............................... 334, 675
get-string-n ................................. 334, 675
get-string-n! ................................ 334, 675
get-u8 ......................................... 330, 674
getaddrinfo .................................. 527
getaffinity .................................... 520
getcwd ......................................... 514
getegid ........................................ 515
getenv ......................................... 513
gteuuid ......................................... 515
gtgid ............................................ 515
gtgr ............................................. 509
gtgrcnt ....................................... 509
gtgrgid ........................................ 508
gtgrnam ........................................ 508
gtgrps .......................................... 514
gethost ........................................ 530
gethostbyaddr ............................... 530
gethostbyname ............................... 530
gethostent ................................... 530
gethostbyname ............................... 542
getitimer ...................................... 523
getlogin ....................................... 509
getnet .......................................... 531
getnetbyaddr ................................. 531
getnetbyname ............................... 531
getnetent ..................................... 531
getopt-long .................................. 575
getpass ....................................... 543
getpeername .................................. 539
getpgid ........................................ 516

getpid ......................................... 514
getppid ........................................ 514
getpriority .................................. 519
getproto ...................................... 532
getprotobyname ............................ 532
getprotobynumber ......................... 532
getprotoent .................................. 532
getpw .......................................... 508
getpwnam ..................................... 508
getpwuid ...................................... 508
getserv ........................................ 533
getservbyname .............................. 533
getservbyport .................................. 533
getservent .................................... 533
getsid ......................................... 516
getsockname .................................. 539
getsockopt .................................... 536
getter-with-setter ....................... 604
ggettext ....................................... 466
gmtime ......................................... 510
guid ............................................ 386
GUILCHECK_RETVAL ............................ 95
GUILFLAGS ..................................... 94
GUIL_MODULE_AVAILABLE .................... 96
GUIL_MODULE_CHECK ......................... 96
GUIL_MODULE_EXPORTS ...................... 96
GUIL_MODULE_REQUIRED ...................... 96
GUIL_MODULE_REQUIRED_EXPORT ........... 96
GUIL_PKG ....................................... 94
GUIL_PROGS .................................... 95
GUIL_SITE_DIR ................................. 95

H

halt ............................................. 829
handle-interrupts ......................... 830
handle-request .............................. 567
hash ........................................... 241, 646
hash-by-identity ........................... 646
hash-clear! ................................... 240
hash-count .................................... 243
hash-create-handle .......................... 242
hash-fold ...................................... 243
hash-for-each ................................ 242
hash-for-each-handle ..................... 242
hash-get-handle ............................. 242
hash-map>list ................................. 242
hash-ref ...................................... 241
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash-remove!</td>
<td>241</td>
</tr>
<tr>
<td>hash-set!</td>
<td>241</td>
</tr>
<tr>
<td>hash-table-alist</td>
<td>646</td>
</tr>
<tr>
<td>hash-table-delete!</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-equivalence-function</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-exists?</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-fold</td>
<td>646</td>
</tr>
<tr>
<td>hash-table-hash-function</td>
<td>646</td>
</tr>
<tr>
<td>hash-table-keys</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-ref</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-ref/default</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-set!</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-size</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-update!</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-update!/default</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-values</td>
<td>645</td>
</tr>
<tr>
<td>hash-table-walk</td>
<td>645</td>
</tr>
<tr>
<td>hash?</td>
<td>240</td>
</tr>
<tr>
<td>hashq-create-handle!</td>
<td>242</td>
</tr>
<tr>
<td>hashq-get-handle</td>
<td>242</td>
</tr>
<tr>
<td>hashq-ref</td>
<td>241</td>
</tr>
<tr>
<td>hashq-remove!</td>
<td>241</td>
</tr>
<tr>
<td>hashq-set!</td>
<td>241</td>
</tr>
<tr>
<td>hashtable-clear!</td>
<td>688</td>
</tr>
<tr>
<td>hashtable-contains?</td>
<td>687</td>
</tr>
<tr>
<td>hashtable-copy</td>
<td>687</td>
</tr>
<tr>
<td>hashtable-delete!</td>
<td>687</td>
</tr>
<tr>
<td>hashtable-entries</td>
<td>688</td>
</tr>
<tr>
<td>hashtable-equivalence-function</td>
<td>688</td>
</tr>
<tr>
<td>hashtable-hash-function</td>
<td>688</td>
</tr>
<tr>
<td>hashtable-keys</td>
<td>688</td>
</tr>
<tr>
<td>hashtable-mutable?</td>
<td>688</td>
</tr>
<tr>
<td>hashtable-ref</td>
<td>687</td>
</tr>
<tr>
<td>hashtable-set!</td>
<td>687</td>
</tr>
<tr>
<td>hashtable-size</td>
<td>687</td>
</tr>
<tr>
<td>hashtable-update!</td>
<td>687</td>
</tr>
<tr>
<td>hashtable?</td>
<td>687</td>
</tr>
<tr>
<td>hashv</td>
<td>241</td>
</tr>
<tr>
<td>hashv-create-handle!</td>
<td>242</td>
</tr>
<tr>
<td>hashv-get-handle</td>
<td>242</td>
</tr>
<tr>
<td>hashv-ref</td>
<td>241</td>
</tr>
<tr>
<td>hashv-remove!</td>
<td>241</td>
</tr>
<tr>
<td>hashv-set!</td>
<td>241</td>
</tr>
<tr>
<td>hashx-create-handle!</td>
<td>242</td>
</tr>
<tr>
<td>hashx-get-handle</td>
<td>242</td>
</tr>
<tr>
<td>hashx-ref</td>
<td>241</td>
</tr>
<tr>
<td>hashx-remove!</td>
<td>241</td>
</tr>
<tr>
<td>hashx-set!</td>
<td>241</td>
</tr>
<tr>
<td>header-&gt;string</td>
<td>548</td>
</tr>
<tr>
<td>header-parser</td>
<td>549</td>
</tr>
<tr>
<td>header-validator</td>
<td>549</td>
</tr>
<tr>
<td>header-writer</td>
<td>549</td>
</tr>
<tr>
<td>help</td>
<td>49</td>
</tr>
<tr>
<td>hook-&gt;list</td>
<td>290</td>
</tr>
<tr>
<td>hook-empty?</td>
<td>290</td>
</tr>
<tr>
<td>hook?</td>
<td>290</td>
</tr>
<tr>
<td>hostent:addr-list</td>
<td>530</td>
</tr>
</tbody>
</table>

**I**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>i/o-encoding-error</td>
<td>671</td>
</tr>
<tr>
<td>i/o-encoding-error-char</td>
<td>671</td>
</tr>
<tr>
<td>i/o-error</td>
<td>671</td>
</tr>
<tr>
<td>i/o-error-char</td>
<td>671</td>
</tr>
<tr>
<td>i/o-error-file</td>
<td>669</td>
</tr>
<tr>
<td>i/o-error-position</td>
<td>669</td>
</tr>
<tr>
<td>i/o-error-port</td>
<td>669</td>
</tr>
<tr>
<td>i/o-file-already-exists</td>
<td>669</td>
</tr>
<tr>
<td>i/o-file-does-not-exist</td>
<td>669</td>
</tr>
<tr>
<td>i/o-file-is-read-only</td>
<td>669</td>
</tr>
<tr>
<td>i/o-file-protection</td>
<td>669</td>
</tr>
<tr>
<td>i/o-file-error</td>
<td>669</td>
</tr>
<tr>
<td>i/o-invalid-name</td>
<td>669</td>
</tr>
<tr>
<td>i/o-invalid-position</td>
<td>669</td>
</tr>
<tr>
<td>i/o-port-error</td>
<td>669</td>
</tr>
<tr>
<td>i/o-read-error</td>
<td>668</td>
</tr>
<tr>
<td>i/o-write-error</td>
<td>669</td>
</tr>
<tr>
<td>i/o-error-port</td>
<td>669</td>
</tr>
<tr>
<td>imax</td>
<td>50</td>
</tr>
<tr>
<td>in</td>
<td>50</td>
</tr>
<tr>
<td>include</td>
<td>394</td>
</tr>
<tr>
<td>include-from-path</td>
<td>395</td>
</tr>
<tr>
<td>inet-lnaof</td>
<td>526</td>
</tr>
<tr>
<td>inet-makeaddr</td>
<td>526</td>
</tr>
<tr>
<td>inet-netof</td>
<td>526</td>
</tr>
<tr>
<td>inet-ntop</td>
<td>526</td>
</tr>
<tr>
<td>inet-pton</td>
<td>527</td>
</tr>
<tr>
<td>inexact</td>
<td>653</td>
</tr>
<tr>
<td>inexact-&gt;exact</td>
<td>114</td>
</tr>
<tr>
<td>inexact?</td>
<td>114</td>
</tr>
<tr>
<td>infinite?</td>
<td>656</td>
</tr>
<tr>
<td>input-port?</td>
<td>329</td>
</tr>
<tr>
<td>inspect</td>
<td>52</td>
</tr>
<tr>
<td>install-suspendable-ports</td>
<td>354</td>
</tr>
<tr>
<td>install-trap-handler</td>
<td>488</td>
</tr>
<tr>
<td>instance?</td>
<td>777</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>instrumented-source-files</td>
<td>490</td>
</tr>
<tr>
<td>instrumented/executed-lines</td>
<td>490</td>
</tr>
<tr>
<td>integer-&gt;char</td>
<td>133, 651, 832</td>
</tr>
<tr>
<td>integer-&gt;list</td>
<td>643</td>
</tr>
<tr>
<td>integer-&gt;expt</td>
<td>127</td>
</tr>
<tr>
<td>integer-length</td>
<td>127</td>
</tr>
<tr>
<td>integer-valued?</td>
<td>656</td>
</tr>
<tr>
<td>integer?</td>
<td>107, 653</td>
</tr>
<tr>
<td>interaction-environment</td>
<td>384</td>
</tr>
<tr>
<td>iota</td>
<td>579</td>
</tr>
<tr>
<td>irritants-condition?</td>
<td>667</td>
</tr>
<tr>
<td>is-a?</td>
<td>777</td>
</tr>
<tr>
<td>isatty?</td>
<td>523</td>
</tr>
<tr>
<td>join-thread</td>
<td>437</td>
</tr>
<tr>
<td>join-timeout-exception?</td>
<td>608</td>
</tr>
<tr>
<td>julian-day-&gt;date</td>
<td>612</td>
</tr>
<tr>
<td>julian-day-&gt;time-monotonic</td>
<td>612</td>
</tr>
<tr>
<td>julian-day-&gt;time-tai</td>
<td>612</td>
</tr>
<tr>
<td>julian-day-&gt;time-utc</td>
<td>612</td>
</tr>
<tr>
<td>keyword-&gt;string</td>
<td>647</td>
</tr>
<tr>
<td>keyword-&gt;symbol</td>
<td>176</td>
</tr>
<tr>
<td>keyword-flatten</td>
<td>370</td>
</tr>
<tr>
<td>keyword?</td>
<td>176, 647</td>
</tr>
<tr>
<td>kill</td>
<td>520</td>
</tr>
<tr>
<td>known-header?</td>
<td>549</td>
</tr>
<tr>
<td>lalr-parser</td>
<td>362</td>
</tr>
<tr>
<td>lambda</td>
<td>249, 652</td>
</tr>
<tr>
<td>lambda*</td>
<td>253</td>
</tr>
<tr>
<td>language</td>
<td>50</td>
</tr>
<tr>
<td>last</td>
<td>581</td>
</tr>
<tr>
<td>last-pair</td>
<td>183</td>
</tr>
<tr>
<td>latin-1-codec</td>
<td>670</td>
</tr>
<tr>
<td>lazy</td>
<td>640</td>
</tr>
<tr>
<td>lchown</td>
<td>503</td>
</tr>
<tr>
<td>lcm</td>
<td>117, 654</td>
</tr>
<tr>
<td>least-fixnum</td>
<td>679</td>
</tr>
<tr>
<td>left-justify-string</td>
<td>753</td>
</tr>
<tr>
<td>length</td>
<td>183, 654</td>
</tr>
<tr>
<td>length*</td>
<td>581</td>
</tr>
<tr>
<td>let</td>
<td>295, 303, 652</td>
</tr>
<tr>
<td>let*</td>
<td>295, 652</td>
</tr>
<tr>
<td>let**-values</td>
<td>603, 652</td>
</tr>
<tr>
<td>let-escape-continuation</td>
<td>305</td>
</tr>
<tr>
<td>let-keywords</td>
<td>255</td>
</tr>
<tr>
<td>let-keywords*</td>
<td>255</td>
</tr>
<tr>
<td>let-optional</td>
<td>254</td>
</tr>
<tr>
<td>let-optional*</td>
<td>254</td>
</tr>
<tr>
<td>let-syntax</td>
<td>262, 652</td>
</tr>
<tr>
<td>let-values</td>
<td>603, 652</td>
</tr>
<tr>
<td>Primitive</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
</tr>
<tr>
<td>primitive-eval</td>
<td>386</td>
</tr>
<tr>
<td>primitive-exit</td>
<td>518</td>
</tr>
<tr>
<td>primitive-fork</td>
<td>519</td>
</tr>
<tr>
<td>primitive-generic-gener ic</td>
<td>768</td>
</tr>
<tr>
<td>primitive-load</td>
<td>389</td>
</tr>
<tr>
<td>primitive-load-path</td>
<td>390</td>
</tr>
<tr>
<td>primitive-move&gt;fdes</td>
<td>405</td>
</tr>
<tr>
<td>print-options</td>
<td>393</td>
</tr>
<tr>
<td>print-set!</td>
<td>394</td>
</tr>
<tr>
<td>procedure</td>
<td>260</td>
</tr>
<tr>
<td>procedure-&gt;pointer</td>
<td>435</td>
</tr>
<tr>
<td>procedure-documentation</td>
<td>259</td>
</tr>
<tr>
<td>procedure-execution-count</td>
<td>490</td>
</tr>
<tr>
<td>procedure-name</td>
<td>259</td>
</tr>
<tr>
<td>procedure-properties</td>
<td>259</td>
</tr>
<tr>
<td>procedure-property</td>
<td>259</td>
</tr>
<tr>
<td>procedure-source</td>
<td>259</td>
</tr>
<tr>
<td>procedure-with-setter?</td>
<td>260</td>
</tr>
<tr>
<td>procedure?</td>
<td>258, 652</td>
</tr>
<tr>
<td>profile</td>
<td>51</td>
</tr>
<tr>
<td>program-arguments</td>
<td>512</td>
</tr>
<tr>
<td>program-arguments-alist</td>
<td>252</td>
</tr>
<tr>
<td>program-arities</td>
<td>251</td>
</tr>
<tr>
<td>program-arity</td>
<td>251</td>
</tr>
<tr>
<td>program-bindings</td>
<td>251</td>
</tr>
<tr>
<td>program-code</td>
<td>250</td>
</tr>
<tr>
<td>program-free-variable-ref</td>
<td>250</td>
</tr>
<tr>
<td>program-free-variable-set!</td>
<td>250</td>
</tr>
<tr>
<td>program-lambda-list</td>
<td>252</td>
</tr>
<tr>
<td>program-num-free-variable</td>
<td>250</td>
</tr>
<tr>
<td>program-sources</td>
<td>251</td>
</tr>
<tr>
<td>program?</td>
<td>250</td>
</tr>
<tr>
<td>promise?</td>
<td>393, 640</td>
</tr>
<tr>
<td>prompt</td>
<td>828</td>
</tr>
<tr>
<td>proper-list?</td>
<td>579</td>
</tr>
<tr>
<td>protoent:aliases</td>
<td>532</td>
</tr>
<tr>
<td>protoent:name</td>
<td>531</td>
</tr>
<tr>
<td>protoent:proto</td>
<td>532</td>
</tr>
<tr>
<td>provide</td>
<td>452</td>
</tr>
<tr>
<td>provided?</td>
<td>452</td>
</tr>
<tr>
<td>PTR2SCM</td>
<td>810</td>
</tr>
<tr>
<td>push</td>
<td>830</td>
</tr>
<tr>
<td>push-dynamic-state</td>
<td>829</td>
</tr>
<tr>
<td>push-fluid</td>
<td>829</td>
</tr>
<tr>
<td>put-bytevector</td>
<td>331, 675</td>
</tr>
<tr>
<td>put-char</td>
<td>334, 675</td>
</tr>
<tr>
<td>put datum</td>
<td>675</td>
</tr>
<tr>
<td>put-string</td>
<td>335, 675</td>
</tr>
<tr>
<td>put-u8</td>
<td>331, 675</td>
</tr>
<tr>
<td>putenv</td>
<td>514</td>
</tr>
<tr>
<td>pwd</td>
<td>514</td>
</tr>
</tbody>
</table>

**Q**
- q-empty-check: 717
- q-empty?: 717
- q-front: 717
- q-length: 717
- q-pop!: 717
- q-push!: 717
- q-rear: 717
- q-remove!: 717
- q?: 716
- quasiquote: 380, 652
- quasiremote: 686
- quit: 53, 518
- quos: 833
- quote: 379, 652
- quotient: 116, 691

**R**
- raise: 521, 608, 665
- raise-continueable: 665
- random: 127
- random-integer: 617
- random-real: 617
- random-source-make-integers: 618
- random-source-make-reals: 618
- random-source-pseudo-randomize!: 618
- random-source-randomize!: 617
- random-source-state-ref: 618
- random-source-state-set!: 618
- random-source??: 617
- random-state->datum: 128
- random-state-from-platform: 129
- random:exp: 128
- random:empty-sphere!: 128
- random:normal: 128
- random:vector!: 128
- random:solid-sphere!: 128
- random:uniform: 128
- rational-valued?: 656
- rational?: 112, 653
- rationalize: 112, 653
- re-export: 412
- read: 382, 678
- read-char: 348, 678
- read-client: 567
- read-delimited: 339
- read-delimited!: 339
- read-disable: 382
- read-enable: 382
- read-hash-extend: 382
- read-header: 550
- read-headers: 550
- read-line: 338
- read-line!: 339
- read-options: 382
- read-request: 560
- read-request-body: 561
<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>read-request-line</td>
<td>550</td>
</tr>
<tr>
<td>read-response</td>
<td>562</td>
</tr>
<tr>
<td>read-response-body</td>
<td>563</td>
</tr>
<tr>
<td>read-response-line</td>
<td>551</td>
</tr>
<tr>
<td>read-set</td>
<td>382</td>
</tr>
<tr>
<td>read-string</td>
<td>748</td>
</tr>
<tr>
<td>read-text-line</td>
<td>748</td>
</tr>
<tr>
<td>read-with-shared-structure</td>
<td>624</td>
</tr>
<tr>
<td>readdir</td>
<td>505</td>
</tr>
<tr>
<td>readline</td>
<td>697</td>
</tr>
<tr>
<td>readline-disable</td>
<td>696</td>
</tr>
<tr>
<td>readline-enable</td>
<td>696</td>
</tr>
<tr>
<td>readline-options</td>
<td>696</td>
</tr>
<tr>
<td>readline-port</td>
<td>697</td>
</tr>
<tr>
<td>readline-set!</td>
<td>696</td>
</tr>
<tr>
<td>readlink</td>
<td>503</td>
</tr>
<tr>
<td>real-&gt;flonum</td>
<td>682</td>
</tr>
<tr>
<td>real-part</td>
<td>118, 653</td>
</tr>
<tr>
<td>real-valued?</td>
<td>656</td>
</tr>
<tr>
<td>real?</td>
<td>111, 653</td>
</tr>
<tr>
<td>rec</td>
<td>619</td>
</tr>
<tr>
<td>receive</td>
<td>310, 823</td>
</tr>
<tr>
<td>receive-values</td>
<td>823</td>
</tr>
<tr>
<td>record-accessor</td>
<td>222, 664</td>
</tr>
<tr>
<td>record-constructor</td>
<td>221, 663</td>
</tr>
<tr>
<td>record-constructor-descriptor</td>
<td>663</td>
</tr>
<tr>
<td>record-field-mutable?</td>
<td>664</td>
</tr>
<tr>
<td>record-modifier</td>
<td>222</td>
</tr>
<tr>
<td>record-mutator</td>
<td>664</td>
</tr>
<tr>
<td>record-predicate</td>
<td>222, 664</td>
</tr>
<tr>
<td>record-rtd</td>
<td>664</td>
</tr>
<tr>
<td>record-type-descriptor</td>
<td>222, 663</td>
</tr>
<tr>
<td>record-type-descriptor?</td>
<td>663</td>
</tr>
<tr>
<td>record-type-field-names</td>
<td>664</td>
</tr>
<tr>
<td>record-type-fields</td>
<td>222</td>
</tr>
<tr>
<td>record-type-generative?</td>
<td>664</td>
</tr>
<tr>
<td>record-type-name</td>
<td>222, 664</td>
</tr>
<tr>
<td>record-type-opaque?</td>
<td>664</td>
</tr>
<tr>
<td>record-type-parent</td>
<td>664</td>
</tr>
<tr>
<td>record-type-sealed?</td>
<td>664</td>
</tr>
<tr>
<td>record-type-uid</td>
<td>664</td>
</tr>
<tr>
<td>record?</td>
<td>221, 664</td>
</tr>
<tr>
<td>recv!</td>
<td>539</td>
</tr>
<tr>
<td>recvfrom!</td>
<td>540</td>
</tr>
<tr>
<td>redirect-port</td>
<td>497</td>
</tr>
<tr>
<td>reduce</td>
<td>583</td>
</tr>
<tr>
<td>reduce-right</td>
<td>583</td>
</tr>
<tr>
<td>regexp-exec</td>
<td>357</td>
</tr>
<tr>
<td>regexp-match</td>
<td>360</td>
</tr>
<tr>
<td>regexp-quote</td>
<td>361</td>
</tr>
<tr>
<td>regexp-substitute</td>
<td>358</td>
</tr>
<tr>
<td>regexp-substitute/global</td>
<td>359</td>
</tr>
<tr>
<td>regexp?</td>
<td>358</td>
</tr>
<tr>
<td>registers</td>
<td>52</td>
</tr>
<tr>
<td>relative-ref</td>
<td>548</td>
</tr>
<tr>
<td>release-port-handle</td>
<td>495</td>
</tr>
<tr>
<td>reload</td>
<td>50</td>
</tr>
<tr>
<td>reload-module</td>
<td>417</td>
</tr>
<tr>
<td>rem</td>
<td>833</td>
</tr>
<tr>
<td>remainder</td>
<td>116, 691</td>
</tr>
<tr>
<td>remove</td>
<td>585, 659</td>
</tr>
<tr>
<td>remove!</td>
<td>585</td>
</tr>
<tr>
<td>remove-class-accessors!</td>
<td>796</td>
</tr>
<tr>
<td>remove-fdes-finalizer!</td>
<td>500</td>
</tr>
<tr>
<td>remove-hook!</td>
<td>290</td>
</tr>
<tr>
<td>remp</td>
<td>659</td>
</tr>
<tr>
<td>remq</td>
<td>659</td>
</tr>
<tr>
<td>rename</td>
<td>504</td>
</tr>
<tr>
<td>rename-file</td>
<td>504</td>
</tr>
<tr>
<td>repl-default-option-set!</td>
<td>53</td>
</tr>
<tr>
<td>replace-range</td>
<td>740</td>
</tr>
<tr>
<td>replace-titles</td>
<td>751</td>
</tr>
<tr>
<td>request-absolute-uri</td>
<td>562</td>
</tr>
<tr>
<td>request-accept</td>
<td>561</td>
</tr>
<tr>
<td>request-accept-charset</td>
<td>561</td>
</tr>
<tr>
<td>request-accept-encoding</td>
<td>561</td>
</tr>
<tr>
<td>request-accept-language</td>
<td>561</td>
</tr>
<tr>
<td>request-allow</td>
<td>561</td>
</tr>
<tr>
<td>request-authorization</td>
<td>561</td>
</tr>
<tr>
<td>request-cache-control</td>
<td>561</td>
</tr>
<tr>
<td>request-connection</td>
<td>561</td>
</tr>
<tr>
<td>request-content-encoding</td>
<td>561</td>
</tr>
<tr>
<td>request-content-language</td>
<td>561</td>
</tr>
<tr>
<td>request-content-length</td>
<td>561</td>
</tr>
<tr>
<td>request-content-location</td>
<td>561</td>
</tr>
<tr>
<td>request-content-md5</td>
<td>561</td>
</tr>
<tr>
<td>request-content-range</td>
<td>561</td>
</tr>
<tr>
<td>request-content-type</td>
<td>561</td>
</tr>
<tr>
<td>request-date</td>
<td>561</td>
</tr>
<tr>
<td>request-expect</td>
<td>561</td>
</tr>
<tr>
<td>request-expires</td>
<td>561</td>
</tr>
<tr>
<td>request-from</td>
<td>561</td>
</tr>
<tr>
<td>request-headers</td>
<td>560</td>
</tr>
<tr>
<td>request-host</td>
<td>561</td>
</tr>
<tr>
<td>request-if-match</td>
<td>561</td>
</tr>
<tr>
<td>request-if-modified-since</td>
<td>561</td>
</tr>
<tr>
<td>request-if-none-match</td>
<td>561</td>
</tr>
<tr>
<td>request-if-range</td>
<td>561</td>
</tr>
<tr>
<td>request-if-unmodified-since</td>
<td>561</td>
</tr>
<tr>
<td>request-last-modified</td>
<td>561</td>
</tr>
<tr>
<td>request-max-forwards</td>
<td>561</td>
</tr>
<tr>
<td>request-meta</td>
<td>560</td>
</tr>
<tr>
<td>request-method</td>
<td>560</td>
</tr>
<tr>
<td>request-port</td>
<td>560</td>
</tr>
<tr>
<td>request-pragma</td>
<td>561</td>
</tr>
<tr>
<td>request-proxy-authorization</td>
<td>561</td>
</tr>
<tr>
<td>request-range</td>
<td>561</td>
</tr>
<tr>
<td>request-referrer</td>
<td>561</td>
</tr>
<tr>
<td>request-te</td>
<td>561</td>
</tr>
<tr>
<td>request-trailer</td>
<td>561</td>
</tr>
<tr>
<td>request-transfer-encoding</td>
<td>561</td>
</tr>
<tr>
<td>request-upgrade</td>
<td>561</td>
</tr>
<tr>
<td>request-uri</td>
<td>560</td>
</tr>
<tr>
<td>request-user-agent</td>
<td>561</td>
</tr>
<tr>
<td>request-version</td>
<td>560</td>
</tr>
</tbody>
</table>
reverse-vector->list 639

S

s16vector 595
s16vector->list 597
s16vector-length 595
s16vector-ref 596
s16vector-set! 597
s16vector? 594
s32vector 595
s32vector->list 597
s32vector-length 595
s32vector-ref 596
s32vector-set! 597
s32vector? 594
s64->scm 835
s64vector 595
s64vector->list 597
s64vector-length 596
s64vector-ref 596
s64vector-set! 597
s64vector? 594
sanitize-response 567
save-module-excursion 417
scandir 714
scheme-report-environment 421, 691
scm->f64 836
scm->pointer 429
scm->s64 835
scm->u64 834
scm->u64/truncate 835
scm-error 317
scm_abs 120
scm_accept 538
scm_access 500
scm_acons 232
scm_add_feature 452
scm_add_hook_x 290

rewinddir 505
rewrite-cont 849
rewrite-exp 849
rewrite-term 849
right-justify-string 753
rmdir 504
rotate-bit-field 643
round! 120, 655
round-ash 126
round-quotient 123
round-remainder 123
round/ 123
run-hook 291
run-server 400, 568

require-extension 641
reset-frame 824
reset-hook! 290
reset-parsed-entity-definitions! 739
resolve 821
resolve-interface 417
resolve-module 417
response-accept-ranges 563
response-age 563
response-allow 563
response-body-port 563
response-cache-control 563
response-code 562
response-connection 563
response-content-encoding 563
response-content-language 563
response-content-length 563
response-content-md5 563
response-content-range 563
response-content-type 563
response-date 563
response-etag 563
response-expires 563
response-headers 562
response-last-modified 563
response-location 563
response-must-not-include-body? 563
response-port 562
response-pragmas 563
response-proxy-authenticate 563
response-reason-phrase 562
responseretry-after 563
response-server 563
response-trailer 563
response-transfer-encoding 564
response-upload 564
response-vary 564
response-version 562
response-via 564
response-warning 564
response-ww-authenticate 564
response? 562
restore-signals 522
restricted-vector-sort! 288
return-from-interrupt 830
return-values 823
reverse 184, 654
reverse! 184
reverse-bit-field 643
reverse-list->string 144
reverse-list->vector 639
reverse-vector->list 639
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>scm_char_set</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_adjoin</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_adjoin_x</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_any</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_complement</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_complement_x</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_contains_p</td>
<td>137</td>
</tr>
<tr>
<td>scm_char_set_copy</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_count</td>
<td>137</td>
</tr>
<tr>
<td>scm_char_set_cursor</td>
<td>134</td>
</tr>
<tr>
<td>scm_char_set_cursor_next</td>
<td>134</td>
</tr>
<tr>
<td>scm_char_set_delete</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_delete_x</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_diff_plus_intersection</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_diff_plus_intersection_x</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_difference</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_difference_x</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_eq</td>
<td>134</td>
</tr>
<tr>
<td>scm_char_set_every</td>
<td>137</td>
</tr>
<tr>
<td>scm_char_set_filter</td>
<td>136</td>
</tr>
<tr>
<td>scm_char_set_filter_x</td>
<td>136</td>
</tr>
<tr>
<td>scm_char_set_fold</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_fold_x</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_for_each</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_hash</td>
<td>134</td>
</tr>
<tr>
<td>scm_char_set_intersection</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_intersection_x</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_map</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_p</td>
<td>134</td>
</tr>
<tr>
<td>scm_char_set_ref</td>
<td>134</td>
</tr>
<tr>
<td>scm_char_set_size</td>
<td>137</td>
</tr>
<tr>
<td>scm_char_set_to_list</td>
<td>137</td>
</tr>
<tr>
<td>scm_char_set_to_string</td>
<td>137</td>
</tr>
<tr>
<td>scm_char_set_unfold</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_unfold_x</td>
<td>135</td>
</tr>
<tr>
<td>scm_char_set_union</td>
<td>138</td>
</tr>
<tr>
<td>scm_char_set_union_x</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_xor</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_set_xor_x</td>
<td>139</td>
</tr>
<tr>
<td>scm_char_titlecase</td>
<td>133</td>
</tr>
<tr>
<td>scm_char_to_integer</td>
<td>133</td>
</tr>
<tr>
<td>scm_char_upcase</td>
<td>133</td>
</tr>
<tr>
<td>scm_char_upper_case_p</td>
<td>132</td>
</tr>
<tr>
<td>scm_char_whitespace_p</td>
<td>132</td>
</tr>
<tr>
<td>scm_chdir</td>
<td>514</td>
</tr>
<tr>
<td>scm_chmod</td>
<td>503</td>
</tr>
<tr>
<td>scm_chown</td>
<td>503</td>
</tr>
<tr>
<td>scm_chroot</td>
<td>514</td>
</tr>
<tr>
<td>scm_close</td>
<td>496</td>
</tr>
<tr>
<td>scm_close_fd</td>
<td>496</td>
</tr>
<tr>
<td>scm_close_port</td>
<td>329</td>
</tr>
<tr>
<td>scm_closedir</td>
<td>505</td>
</tr>
<tr>
<td>scm_complex_p</td>
<td>113</td>
</tr>
<tr>
<td>scm_condition_variable_p</td>
<td>445</td>
</tr>
<tr>
<td>scm_connect</td>
<td>537</td>
</tr>
<tr>
<td>scm_cons</td>
<td>179</td>
</tr>
<tr>
<td>scm_cons_source</td>
<td>472</td>
</tr>
<tr>
<td>scm_copy_file</td>
<td>503</td>
</tr>
<tr>
<td>scm_make_guardian</td>
<td>406</td>
</tr>
<tr>
<td>scm_make_book</td>
<td>290</td>
</tr>
<tr>
<td>scm_make_locale</td>
<td>459</td>
</tr>
<tr>
<td>scm_make_mutex</td>
<td>443</td>
</tr>
<tr>
<td>scm_make_mutex_with_kind</td>
<td>443</td>
</tr>
<tr>
<td>scm_make_polar</td>
<td>118</td>
</tr>
<tr>
<td>scm_make_port_type</td>
<td>350</td>
</tr>
<tr>
<td>scm_make_procedure_with_setter</td>
<td>260</td>
</tr>
<tr>
<td>scm_make_rectangle</td>
<td>118</td>
</tr>
<tr>
<td>scm_make_recursive_mutex</td>
<td>444</td>
</tr>
<tr>
<td>scm_make_regexp</td>
<td>356</td>
</tr>
<tr>
<td>scm_make_s16vector</td>
<td>595</td>
</tr>
<tr>
<td>scm_make_s32vector</td>
<td>595</td>
</tr>
<tr>
<td>scm_make_s64vector</td>
<td>595</td>
</tr>
<tr>
<td>scm_make_s8vector</td>
<td>594</td>
</tr>
<tr>
<td>scm_make_shared_array</td>
<td>206</td>
</tr>
<tr>
<td>scm_make_smob_type</td>
<td>245</td>
</tr>
<tr>
<td>scm_make_socket_address</td>
<td>534</td>
</tr>
<tr>
<td>scm_make_stack</td>
<td>468</td>
</tr>
<tr>
<td>scm_make_string</td>
<td>144</td>
</tr>
<tr>
<td>scm_make_struct</td>
<td>224</td>
</tr>
<tr>
<td>scm_make_struct_layout</td>
<td>227</td>
</tr>
<tr>
<td>scm_make_symbol</td>
<td>173</td>
</tr>
<tr>
<td>scm_make_thread_local_fluid</td>
<td>438</td>
</tr>
<tr>
<td>scm_make.Typed_array</td>
<td>203</td>
</tr>
<tr>
<td>scm_make_u16vector</td>
<td>595</td>
</tr>
<tr>
<td>scm_make_u32vector</td>
<td>595</td>
</tr>
<tr>
<td>scm_make_u64vector</td>
<td>595</td>
</tr>
<tr>
<td>scm_make_u8vector</td>
<td>594</td>
</tr>
<tr>
<td>scm_make.unbound_fluid</td>
<td>321</td>
</tr>
<tr>
<td>scm_make.undefined_variable</td>
<td>416</td>
</tr>
<tr>
<td>scm_make_variable</td>
<td>416</td>
</tr>
<tr>
<td>scm_make_vector</td>
<td>187</td>
</tr>
<tr>
<td>scm_make_weak_key_hash_table</td>
<td>404</td>
</tr>
<tr>
<td>scm_make_weak_value_hash_table</td>
<td>404</td>
</tr>
<tr>
<td>scm_make_weak_vector</td>
<td>405</td>
</tr>
<tr>
<td>scm_malloc</td>
<td>402</td>
</tr>
<tr>
<td>scm_map</td>
<td>186</td>
</tr>
<tr>
<td>scm_markcdr</td>
<td>248</td>
</tr>
<tr>
<td>scm_max</td>
<td>120</td>
</tr>
<tr>
<td>scm_member</td>
<td>186</td>
</tr>
<tr>
<td>scm_memory_error</td>
<td>327</td>
</tr>
<tr>
<td>scm_memq</td>
<td>185</td>
</tr>
<tr>
<td>scm_memv</td>
<td>186</td>
</tr>
<tr>
<td>scm_merge</td>
<td>286</td>
</tr>
<tr>
<td>scm_merge_x</td>
<td>286</td>
</tr>
<tr>
<td>scm_micro_version</td>
<td>450</td>
</tr>
<tr>
<td>scm_min</td>
<td>120</td>
</tr>
<tr>
<td>scm_minor_version</td>
<td>450</td>
</tr>
<tr>
<td>scm_misc_error</td>
<td>327</td>
</tr>
<tr>
<td>scm_mkdir</td>
<td>504</td>
</tr>
<tr>
<td>scm_mkmod</td>
<td>505</td>
</tr>
<tr>
<td>scm_mkstemp</td>
<td>506</td>
</tr>
<tr>
<td>scm_mktime</td>
<td>511</td>
</tr>
<tr>
<td>scm_module_define</td>
<td>420</td>
</tr>
<tr>
<td>scm_module.ensure_local_variable</td>
<td>420</td>
</tr>
<tr>
<td>scm_module.lookup</td>
<td>419</td>
</tr>
<tr>
<td>scm_module_reverse_lookup</td>
<td>420</td>
</tr>
</tbody>
</table>
tail-apply.......................... 826
tail-call.......................... 823
tail-call-label..................... 823
tail-call/shuffle................... 823
take................................. 580
take!................................ 580
take-after............................ 744
take-right............................ 581
take-until............................ 744
take-while........................... 586
take-while!........................... 586
tan.................................. 124, 653
tanh.................................. 124
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tm:gmtoff</code></td>
<td>510</td>
</tr>
<tr>
<td><code>tm:hour</code></td>
<td>509</td>
</tr>
<tr>
<td><code>tm:isdst</code></td>
<td>510</td>
</tr>
<tr>
<td><code>tm:mon</code></td>
<td>509</td>
</tr>
<tr>
<td><code>tm:sec</code></td>
<td>509</td>
</tr>
<tr>
<td><code>tm:weekday</code></td>
<td>510</td>
</tr>
<tr>
<td><code>tm:year</code></td>
<td>510</td>
</tr>
<tr>
<td><code>tmpfile</code></td>
<td>506</td>
</tr>
<tr>
<td><code>tm:clock</code></td>
<td>512</td>
</tr>
<tr>
<td><code>tm:ctime</code></td>
<td>512</td>
</tr>
<tr>
<td><code>tm:utime</code></td>
<td>512</td>
</tr>
<tr>
<td><code>toplevel-box</code></td>
<td>822</td>
</tr>
<tr>
<td><code>total-processor-count</code></td>
<td>438</td>
</tr>
<tr>
<td><code>trace</code></td>
<td>448</td>
</tr>
<tr>
<td><code>trace-calls-in-procedure</code></td>
<td>486</td>
</tr>
<tr>
<td><code>trace-calls-to-procedure</code></td>
<td>486</td>
</tr>
<tr>
<td><code>trace-instructions-in-procedure</code></td>
<td>486</td>
</tr>
<tr>
<td><code>tracepoint</code></td>
<td>52</td>
</tr>
<tr>
<td><code>transcoded-port</code></td>
<td>673</td>
</tr>
<tr>
<td><code>transcoder-codel</code></td>
<td>672</td>
</tr>
<tr>
<td><code>transcoder-eol-style</code></td>
<td>672</td>
</tr>
<tr>
<td><code>transform-string</code></td>
<td>752</td>
</tr>
<tr>
<td><code>transpose-array</code></td>
<td>207</td>
</tr>
<tr>
<td><code>trap-at-procedure-call</code></td>
<td>483</td>
</tr>
<tr>
<td><code>trap-at-procedure-ip-in-range</code></td>
<td>484</td>
</tr>
<tr>
<td><code>trap-at-source-location</code></td>
<td>484</td>
</tr>
<tr>
<td><code>trap-calls-in-procedure</code></td>
<td>484</td>
</tr>
<tr>
<td><code>trap-calls-to-procedure</code></td>
<td>485</td>
</tr>
<tr>
<td><code>trap-enabled?</code></td>
<td>487</td>
</tr>
<tr>
<td><code>trap-frame-finish</code></td>
<td>484</td>
</tr>
<tr>
<td><code>trap-in-procedure</code></td>
<td>483</td>
</tr>
<tr>
<td><code>trap-instructions-in-procedure</code></td>
<td>484</td>
</tr>
<tr>
<td><code>trap-instructions-in-dynamic-extent</code></td>
<td>484</td>
</tr>
<tr>
<td><code>trap-name</code></td>
<td>487</td>
</tr>
<tr>
<td><code>truncatem</code></td>
<td>120, 338, 655</td>
</tr>
<tr>
<td><code>truncatem-file</code></td>
<td>338</td>
</tr>
<tr>
<td><code>truncatem-quotient</code></td>
<td>122</td>
</tr>
<tr>
<td><code>truncatem-remainder</code></td>
<td>122</td>
</tr>
<tr>
<td><code>truncatem/</code></td>
<td>122</td>
</tr>
<tr>
<td><code>truncated-print</code></td>
<td>700</td>
</tr>
<tr>
<td><code>try-mutex</code></td>
<td>444</td>
</tr>
<tr>
<td><code>ttyname</code></td>
<td>524</td>
</tr>
<tr>
<td><code>typed-array?</code></td>
<td>202</td>
</tr>
<tr>
<td><code>tzset</code></td>
<td>511</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>unread-string</td>
<td>349</td>
</tr>
<tr>
<td>unsetenv</td>
<td>513</td>
</tr>
<tr>
<td>unsyntax</td>
<td>866</td>
</tr>
<tr>
<td>unsyntax-splicing</td>
<td>866</td>
</tr>
<tr>
<td>unwind</td>
<td>829</td>
</tr>
<tr>
<td>unzip1</td>
<td>582</td>
</tr>
<tr>
<td>unzip2</td>
<td>582</td>
</tr>
<tr>
<td>unzip3</td>
<td>582</td>
</tr>
<tr>
<td>unzip4</td>
<td>582</td>
</tr>
<tr>
<td>unzip5</td>
<td>582</td>
</tr>
<tr>
<td>vector</td>
<td>51</td>
</tr>
<tr>
<td>update-direct-method!</td>
<td>796</td>
</tr>
<tr>
<td>update-direct-subclass!</td>
<td>796</td>
</tr>
<tr>
<td>update-instance-for-different-class</td>
<td>797</td>
</tr>
<tr>
<td>uri-&gt;string</td>
<td>546</td>
</tr>
<tr>
<td>uri-decode</td>
<td>546</td>
</tr>
<tr>
<td>uri-encode</td>
<td>547</td>
</tr>
<tr>
<td>uri-fragment</td>
<td>546</td>
</tr>
<tr>
<td>uri-host</td>
<td>546</td>
</tr>
<tr>
<td>uri-path</td>
<td>546</td>
</tr>
<tr>
<td>uri-port</td>
<td>546</td>
</tr>
<tr>
<td>uri-query</td>
<td>546</td>
</tr>
<tr>
<td>uri-reference?</td>
<td>547</td>
</tr>
<tr>
<td>uri-scheme</td>
<td>546</td>
</tr>
<tr>
<td>uri-userinfo</td>
<td>546</td>
</tr>
<tr>
<td>uri?</td>
<td>546</td>
</tr>
<tr>
<td>urlify</td>
<td>751</td>
</tr>
<tr>
<td>ursh</td>
<td>836</td>
</tr>
<tr>
<td>ursh/mediate</td>
<td>836</td>
</tr>
<tr>
<td>use-modules</td>
<td>408</td>
</tr>
<tr>
<td>usleep</td>
<td>522</td>
</tr>
<tr>
<td>usub</td>
<td>835</td>
</tr>
<tr>
<td>usub/mediate</td>
<td>835</td>
</tr>
<tr>
<td>utf-16-codec</td>
<td>670</td>
</tr>
<tr>
<td>utf-8-codec</td>
<td>670</td>
</tr>
<tr>
<td>utf16-&gt;string</td>
<td>199</td>
</tr>
<tr>
<td>utf32-&gt;string</td>
<td>199</td>
</tr>
<tr>
<td>utf8-&gt;string</td>
<td>199</td>
</tr>
<tr>
<td>utime</td>
<td>503</td>
</tr>
<tr>
<td>utnsname:machine</td>
<td>542</td>
</tr>
<tr>
<td>utnsname:nodename</td>
<td>542</td>
</tr>
<tr>
<td>utnsname:release</td>
<td>542</td>
</tr>
<tr>
<td>utnsname:sysname</td>
<td>542</td>
</tr>
<tr>
<td>utnsname:version</td>
<td>542</td>
</tr>
<tr>
<td>vector-apply</td>
<td>638</td>
</tr>
<tr>
<td>vector-append</td>
<td>636</td>
</tr>
<tr>
<td>vector-binary-search</td>
<td>638</td>
</tr>
<tr>
<td>vector-concatenate</td>
<td>636</td>
</tr>
<tr>
<td>vector-copy</td>
<td>189, 645</td>
</tr>
<tr>
<td>vector-copy!</td>
<td>639</td>
</tr>
<tr>
<td>vector-count</td>
<td>637</td>
</tr>
<tr>
<td>vector-empty?</td>
<td>636</td>
</tr>
<tr>
<td>vector-every</td>
<td>638</td>
</tr>
<tr>
<td>vector-fill!</td>
<td>189, 639, 657</td>
</tr>
<tr>
<td>vector-fold</td>
<td>636</td>
</tr>
<tr>
<td>vector-fold-right</td>
<td>637</td>
</tr>
<tr>
<td>vector-for-each</td>
<td>637, 656</td>
</tr>
<tr>
<td>vector-index</td>
<td>637</td>
</tr>
<tr>
<td>vector-index-right</td>
<td>637</td>
</tr>
<tr>
<td>vector-length</td>
<td>188, 636, 657, 830, 853</td>
</tr>
<tr>
<td>vector-map</td>
<td>637, 656</td>
</tr>
<tr>
<td>vector-map!</td>
<td>637</td>
</tr>
<tr>
<td>vector-move-left!</td>
<td>189</td>
</tr>
<tr>
<td>vector-move-right!</td>
<td>189</td>
</tr>
<tr>
<td>vector-ref</td>
<td>188, 636, 657, 830</td>
</tr>
<tr>
<td>vector-ref/immediate</td>
<td>830</td>
</tr>
<tr>
<td>vector-reverse!</td>
<td>639</td>
</tr>
<tr>
<td>vector-reverse-copy</td>
<td>636</td>
</tr>
<tr>
<td>vector-reverse-copy!</td>
<td>639</td>
</tr>
<tr>
<td>vector-set!</td>
<td>188, 636, 657, 831</td>
</tr>
<tr>
<td>vector-set!/immediate</td>
<td>831</td>
</tr>
<tr>
<td>vector-skip</td>
<td>638</td>
</tr>
<tr>
<td>vector-skip-right</td>
<td>638</td>
</tr>
<tr>
<td>vector-sort</td>
<td>660</td>
</tr>
<tr>
<td>vector-sort!</td>
<td>660</td>
</tr>
<tr>
<td>vector-swap</td>
<td>639</td>
</tr>
<tr>
<td>vector-undefined</td>
<td>635</td>
</tr>
<tr>
<td>vector-undefined-right</td>
<td>635</td>
</tr>
<tr>
<td>vector=?</td>
<td>636</td>
</tr>
<tr>
<td>version</td>
<td>450</td>
</tr>
<tr>
<td>vhash-assoc</td>
<td>236</td>
</tr>
<tr>
<td>vhash-assq</td>
<td>236</td>
</tr>
<tr>
<td>vhash-assv</td>
<td>236</td>
</tr>
<tr>
<td>vhash-cons</td>
<td>236</td>
</tr>
<tr>
<td>vhash-consq</td>
<td>236</td>
</tr>
<tr>
<td>vhash-consv</td>
<td>236</td>
</tr>
<tr>
<td>vhash-delete</td>
<td>237</td>
</tr>
<tr>
<td>vhash-delq</td>
<td>237</td>
</tr>
<tr>
<td>vhash-delv</td>
<td>237</td>
</tr>
<tr>
<td>vhash-fold</td>
<td>237</td>
</tr>
<tr>
<td>vhash-fold*</td>
<td>237</td>
</tr>
<tr>
<td>vhash-fold-right</td>
<td>237</td>
</tr>
<tr>
<td>vhash-foldq*</td>
<td>237</td>
</tr>
<tr>
<td>vhash-foldv*</td>
<td>237</td>
</tr>
<tr>
<td>vhash?</td>
<td>236</td>
</tr>
<tr>
<td>violation?</td>
<td>667</td>
</tr>
<tr>
<td>vlist-&gt;list</td>
<td>217</td>
</tr>
<tr>
<td>vlist-append</td>
<td>217</td>
</tr>
<tr>
<td>vlist-cons</td>
<td>216</td>
</tr>
<tr>
<td>vlist-delete</td>
<td>217</td>
</tr>
<tr>
<td>vlist-drop</td>
<td>217</td>
</tr>
</tbody>
</table>
vlist-filter ........................................... 217
vlist-fold ............................................. 216
vlist-fold-right ..................................... 216
vlist-for-each ........................................ 217
vlist-head ............................................ 216
vlist-length ......................................... 217
vlist-map .............................................. 217
vlist-null? ............................................ 216
vlist-ref ............................................... 217
vlist-reverse ........................................ 217
vlist-tail ............................................. 216
vlist-take ............................................ 217
vlist-unfold ......................................... 217
vlist-unfold-right .................................. 217
vlist? .................................................. 216
vm-abort-continuation-hook ....................... 482
vm-apply-hook ....................................... 481
vm-next-hook ........................................ 481
vm-pop-continuation-hook ......................... 481
vm-push-continuation-hook ......................... 481
vm-restore-continuation-hook ...................... 482
vm-trace-level ...................................... 482

W
wait-condition-variable .............................. 445
waitpid ............................................... 516
warning? ............................................. 667
weak-key-hash-table? ................................ 405
weak-value-hash-table? .............................. 405
weak-vector .......................................... 405
weak-vector-ref ...................................... 405
weak-vector-set! ..................................... 405
weak-vector? ........................................ 405
when ................................................... 299, 660
while ................................................... 302
who-condition? ....................................... 667
width ................................................... 52
wind ................................................... 829
with-code-coverage .................................. 489
with-continuation-barrier ......................... 328
with-default-trap-handler ......................... 488
with-dynamic-state .................................. 323
with-ellipsis ........................................ 272
with-error-to-file .................................. 343
with-error-to-port .................................. 340
with-exception-handler ......................... 608, 665
with-fluid* ......................................... 322
with-fluids ......................................... 322
with-fluids* ........................................ 322
with-input-from-file ................................. 343, 678
with-input-from-port ............................... 340
with-input-from-string .............................. 344
with-mutex ........................................... 445
with-output-to-file ................................. 343, 678
with-output-to-port ................................ 340
with-output-to-string ............................... 344
with-parameters* .................................... 625
with-readline-completion-function .............. 698
with-ssax-error-to-port ............................ 738
with-syntax ......................................... 271
with-throw-handler .................................. 314
write ................................................. 383, 678, 783
write-char .......................................... 349, 678
write-client ........................................ 567
write-header ......................................... 550
write-headers ....................................... 550
write-request ....................................... 560
write-request-body .................................. 561
write-request-line .................................. 550
write-response ...................................... 563
write-response-body ................................ 563
write-response-line ................................ 551
write-with-shared-structure ...................... 623

X
xcons .................................................. 579
xml->sxml ............................................ 733
xml-token-head ..................................... 738
xml-token-kind ..................................... 738
xml-token? .......................................... 738
xsubstring .......................................... 158

Y
yield ................................................... 437

Z
zero? ................................................... 117, 654
zip .................................................... 581
Variable Index

This is an alphabetical list of all the important variables and constants in Guile.

When looking for a particular variable or constant, please look under its Scheme name as well as under its C name. The C name can be constructed from the Scheme names by a simple transformation described in the section See Section 6.1 [API Overview], page 99.

<table>
<thead>
<tr>
<th>%</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%auto- compilation- options</td>
<td>388</td>
</tr>
<tr>
<td>%default- port- encoding</td>
<td>331</td>
</tr>
<tr>
<td>%global- locale</td>
<td>460</td>
</tr>
<tr>
<td>%guile- build- info</td>
<td>451</td>
</tr>
<tr>
<td>%host- type</td>
<td>452</td>
</tr>
<tr>
<td>%load- compiled- path</td>
<td>391</td>
</tr>
<tr>
<td>%load- extensions</td>
<td>390</td>
</tr>
<tr>
<td>%load- hook</td>
<td>389</td>
</tr>
<tr>
<td>%load- path</td>
<td>389</td>
</tr>
<tr>
<td>%null- pointer</td>
<td>429</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&amp;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;condition</td>
<td>621</td>
</tr>
<tr>
<td>&amp;error</td>
<td>622</td>
</tr>
<tr>
<td>&amp;message</td>
<td>621</td>
</tr>
<tr>
<td>&amp;serious</td>
<td>621</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>*</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>features</em></td>
<td>452</td>
</tr>
<tr>
<td><em>random- state</em></td>
<td>129</td>
</tr>
<tr>
<td><em>sdocbook-&gt;stexi- rules</em></td>
<td>750</td>
</tr>
<tr>
<td><em>sdocbook- block- commands</em></td>
<td>750</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;standard- vtable&gt;</td>
<td>226</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>555</td>
</tr>
<tr>
<td>accept- charset</td>
<td>555</td>
</tr>
<tr>
<td>accept- encoding</td>
<td>555</td>
</tr>
<tr>
<td>accept- language</td>
<td>555</td>
</tr>
<tr>
<td>accept- ranges</td>
<td>557</td>
</tr>
<tr>
<td>after- gc- hook</td>
<td>293</td>
</tr>
<tr>
<td>age</td>
<td>557</td>
</tr>
<tr>
<td>alist- bindings</td>
<td>398</td>
</tr>
<tr>
<td>all- pure- and- impure- bindings</td>
<td>398</td>
</tr>
<tr>
<td>all- pure- bindings</td>
<td>398</td>
</tr>
<tr>
<td>allow</td>
<td>553</td>
</tr>
<tr>
<td>array- bindings</td>
<td>398</td>
</tr>
<tr>
<td>AT_ SYMLINK_ NOFOLLOW</td>
<td>503</td>
</tr>
<tr>
<td>authorization</td>
<td>555</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bit- bindings</td>
<td>398</td>
</tr>
<tr>
<td>bitvector- bindings</td>
<td>398</td>
</tr>
<tr>
<td>block- growth- factor</td>
<td>216</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cache- control</td>
<td>552</td>
</tr>
<tr>
<td>char- bindings</td>
<td>398</td>
</tr>
<tr>
<td>char- set- bindings</td>
<td>398</td>
</tr>
<tr>
<td>char- set: ascii</td>
<td>140</td>
</tr>
<tr>
<td>char- set: blank</td>
<td>140</td>
</tr>
<tr>
<td>char- set: designated</td>
<td>141</td>
</tr>
<tr>
<td>char- set: digit</td>
<td>140</td>
</tr>
<tr>
<td>char- set: empty</td>
<td>141</td>
</tr>
<tr>
<td>char- set: full</td>
<td>141</td>
</tr>
<tr>
<td>char- set: graphic</td>
<td>140</td>
</tr>
<tr>
<td>char- set: hex- digit</td>
<td>140</td>
</tr>
<tr>
<td>char- set: iso- control</td>
<td>140</td>
</tr>
<tr>
<td>char- set: letter</td>
<td>140</td>
</tr>
<tr>
<td>char- set: letter+ digit</td>
<td>140</td>
</tr>
<tr>
<td>char- set: lower- case</td>
<td>139</td>
</tr>
<tr>
<td>char- set: printing</td>
<td>140</td>
</tr>
<tr>
<td>char- set: punctuation</td>
<td>140</td>
</tr>
<tr>
<td>char- set: symbol</td>
<td>140</td>
</tr>
<tr>
<td>char- set: title- case</td>
<td>139</td>
</tr>
<tr>
<td>char- set: upper- case</td>
<td>139</td>
</tr>
<tr>
<td>char- set: whitespace</td>
<td>140</td>
</tr>
<tr>
<td>clock- bindings</td>
<td>398</td>
</tr>
<tr>
<td>connection</td>
<td>552</td>
</tr>
<tr>
<td>content- encoding</td>
<td>553</td>
</tr>
<tr>
<td>content- language</td>
<td>553</td>
</tr>
<tr>
<td>content- length</td>
<td>554</td>
</tr>
<tr>
<td>content- location</td>
<td>554</td>
</tr>
<tr>
<td>content-md5</td>
<td>554</td>
</tr>
<tr>
<td>content- range</td>
<td>554</td>
</tr>
<tr>
<td>content- type</td>
<td>554</td>
</tr>
<tr>
<td>core- bindings</td>
<td>398</td>
</tr>
<tr>
<td>current- reader</td>
<td>389</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>552</td>
</tr>
<tr>
<td>double</td>
<td>428</td>
</tr>
<tr>
<td>Variable</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>O_APPEND</td>
<td>496</td>
</tr>
<tr>
<td>O_CREAT</td>
<td>496</td>
</tr>
<tr>
<td>O_RDONLY</td>
<td>495</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>496</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>496</td>
</tr>
<tr>
<td>O_WRONLY</td>
<td>496</td>
</tr>
<tr>
<td>OPEN_BOTH</td>
<td>524</td>
</tr>
<tr>
<td>OPEN_READ</td>
<td>524</td>
</tr>
<tr>
<td>OPEN_WRITE</td>
<td>524</td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>pair-bindings</td>
<td>398</td>
</tr>
<tr>
<td>PF_INET</td>
<td>535</td>
</tr>
<tr>
<td>PF_INET6</td>
<td>535</td>
</tr>
<tr>
<td>PF_UNIX</td>
<td>535</td>
</tr>
<tr>
<td>PIPE_BUF</td>
<td>496</td>
</tr>
<tr>
<td>pragma</td>
<td>552</td>
</tr>
<tr>
<td>predicate-bindings</td>
<td>398</td>
</tr>
<tr>
<td>PRIO_PGRP</td>
<td>519</td>
</tr>
<tr>
<td>PRIO_PROCESS</td>
<td>519</td>
</tr>
<tr>
<td>PRIO_USER</td>
<td>519</td>
</tr>
<tr>
<td>procedure-bindings</td>
<td>398</td>
</tr>
<tr>
<td>promise-bindings</td>
<td>398</td>
</tr>
<tr>
<td>prompt-bindings</td>
<td>398</td>
</tr>
<tr>
<td>proxy-authenticate</td>
<td>558</td>
</tr>
<tr>
<td>proxy-authorization</td>
<td>556</td>
</tr>
<tr>
<td>ptdiff_t</td>
<td>428</td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>R_OK</td>
<td>500</td>
</tr>
<tr>
<td>range</td>
<td>557</td>
</tr>
<tr>
<td>referer</td>
<td>557</td>
</tr>
<tr>
<td>regexp-bindings</td>
<td>398</td>
</tr>
<tr>
<td>regexp/basic</td>
<td>357</td>
</tr>
<tr>
<td>regexp/extended</td>
<td>357</td>
</tr>
<tr>
<td>regexp/icase</td>
<td>356</td>
</tr>
<tr>
<td>regexp/newline</td>
<td>357</td>
</tr>
<tr>
<td>regexp/notbol</td>
<td>357</td>
</tr>
<tr>
<td>retry-after</td>
<td>558</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>SA_NOCLDSTOP</td>
<td>521</td>
</tr>
<tr>
<td>SA_RESTART</td>
<td>521</td>
</tr>
<tr>
<td>scm_after_gc_c_hook</td>
<td>293</td>
</tr>
<tr>
<td>scm_after_gc_hook</td>
<td>293</td>
</tr>
<tr>
<td>scm_after_sweep_c_hook</td>
<td>293</td>
</tr>
<tr>
<td>scm_before_gc_c_hook</td>
<td>292</td>
</tr>
<tr>
<td>scm_before_mark_c_hook</td>
<td>293</td>
</tr>
<tr>
<td>scm_before_sweep_c_hook</td>
<td>293</td>
</tr>
<tr>
<td>scm_char_set_ascii</td>
<td>140</td>
</tr>
<tr>
<td>scm_char_set_blank</td>
<td>140</td>
</tr>
<tr>
<td>scm_char_set_designated</td>
<td>141</td>
</tr>
<tr>
<td>scm_char_set_digit</td>
<td>140</td>
</tr>
<tr>
<td>Symbol</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
</tr>
<tr>
<td>SOCK_DGRAM</td>
<td>535</td>
</tr>
<tr>
<td>SOCK_RAW</td>
<td>535</td>
</tr>
<tr>
<td>SOCK_RDM</td>
<td>535</td>
</tr>
<tr>
<td>SOCK_SEQPACKET</td>
<td>535</td>
</tr>
<tr>
<td>SOCK_STREAM</td>
<td>535</td>
</tr>
<tr>
<td>SQL_SOCKET</td>
<td>536</td>
</tr>
<tr>
<td>sort-bindings</td>
<td>398</td>
</tr>
<tr>
<td>srffi-4-bindings</td>
<td>398</td>
</tr>
<tr>
<td>ssize_t</td>
<td>428</td>
</tr>
<tr>
<td>standard-vtable-fields</td>
<td>227</td>
</tr>
<tr>
<td>stream-null</td>
<td>625</td>
</tr>
<tr>
<td>string-bindings</td>
<td>398</td>
</tr>
<tr>
<td>symbol-bindings</td>
<td>398</td>
</tr>
<tr>
<td>TCP_CORK</td>
<td>537</td>
</tr>
<tr>
<td>TCP_NODELAY</td>
<td>537</td>
</tr>
<tr>
<td>te</td>
<td>557</td>
</tr>
<tr>
<td>teixi-command-specs</td>
<td>749</td>
</tr>
<tr>
<td>time-duration</td>
<td>609</td>
</tr>
<tr>
<td>time-monotonic</td>
<td>609</td>
</tr>
<tr>
<td>time-process</td>
<td>609</td>
</tr>
<tr>
<td>time-tai</td>
<td>609</td>
</tr>
<tr>
<td>time-thread</td>
<td>609</td>
</tr>
<tr>
<td>time-utc</td>
<td>609</td>
</tr>
<tr>
<td>trailer</td>
<td>552</td>
</tr>
<tr>
<td>transfer-encoding</td>
<td>553</td>
</tr>
<tr>
<td>uint16</td>
<td>428</td>
</tr>
<tr>
<td>uint32</td>
<td>428</td>
</tr>
<tr>
<td>uint64</td>
<td>428</td>
</tr>
<tr>
<td>uint8</td>
<td>428</td>
</tr>
<tr>
<td>uintptr_t</td>
<td>428</td>
</tr>
<tr>
<td>unsigned-int</td>
<td>428</td>
</tr>
<tr>
<td>unsigned-long</td>
<td>428</td>
</tr>
<tr>
<td>unsigned-short</td>
<td>428</td>
</tr>
<tr>
<td>unspecified-bindings</td>
<td>398</td>
</tr>
<tr>
<td>upgrade</td>
<td>553</td>
</tr>
<tr>
<td>user-agent</td>
<td>557</td>
</tr>
<tr>
<td>variable-bindings</td>
<td>398</td>
</tr>
<tr>
<td>vary</td>
<td>558</td>
</tr>
<tr>
<td>vector-bindings</td>
<td>398</td>
</tr>
<tr>
<td>version-bindings</td>
<td>398</td>
</tr>
<tr>
<td>via</td>
<td>553</td>
</tr>
<tr>
<td>vlist-null</td>
<td>216</td>
</tr>
<tr>
<td>void</td>
<td>428</td>
</tr>
<tr>
<td>vtable-index-layout</td>
<td>225</td>
</tr>
<tr>
<td>vtable-index-printer</td>
<td>225</td>
</tr>
<tr>
<td>vtable-offset-user</td>
<td>227</td>
</tr>
<tr>
<td>w_OK</td>
<td>500</td>
</tr>
<tr>
<td>WAIT_ANY</td>
<td>516</td>
</tr>
<tr>
<td>WAIT_MYPGRP</td>
<td>516</td>
</tr>
<tr>
<td>warning</td>
<td>553</td>
</tr>
<tr>
<td>WNOHANG</td>
<td>517</td>
</tr>
<tr>
<td>WUNTRACED</td>
<td>517</td>
</tr>
<tr>
<td>www-authenticate</td>
<td>558</td>
</tr>
<tr>
<td>X_OK</td>
<td>501</td>
</tr>
</tbody>
</table>
Type Index

This is an alphabetical list of all the important data types defined in the Guile Programmers Manual.

\$  
\$arity 848
\$branch 846
\$call 846
\$callk 847
\$closure 847
\$const 846
\$continue 845
\$fun 847
\$kargs 845
\$kclause 848
\$kfun 848
\$kreceive 848
\$ktail 848
\$prim 846
\$primcall 846
\$prompt 846
\$rec 847
\$values 846

(  
(0 841
(00 841
(abort 843
(call 841
(const 840
(define 841
(if 841
(lambda 842
(lambda-case 842
(let 842
(let-values 843
(letrec 842
(letrec* 842
(lexical 840
(primcall 841
(primitive 840
(prompt 842
(seq 841
(set! 841
(toplevel 841
(void) 840

<  
<abort> 843
<call> 841
<conditional> 841
<const> 840
<fix> 843
<lambda-case> 842
<lambda> 842
<let-values> 843
<let> 842
<letrec> 842
<lexical-ref> 840
<lexical-set> 841
<module-ref> 841
<module-set> 841
<primcall> 841
<primitive-ref> 840
<prompt> 842
<seq> 841
<toplevel-define> 841
<toplevel-ref> 841
<toplevel-set> 841
<void> 840

A  
Alist 229
any character 364
Arrays 200
Association Lists 229

B  
Booleans 104

C  
capture 365
character class 364
Characters 129
Complex numbers 113

D  
Date 551

E  
ETag 552
Exact numbers 113
R5RS Index

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>char-downcase</td>
</tr>
<tr>
<td></td>
<td>char-lower-case?</td>
</tr>
<tr>
<td></td>
<td>char-numeric?</td>
</tr>
<tr>
<td></td>
<td>char-ready?</td>
</tr>
<tr>
<td></td>
<td>char-titlecase</td>
</tr>
<tr>
<td></td>
<td>char-upcase</td>
</tr>
<tr>
<td></td>
<td>char-upper-case?</td>
</tr>
<tr>
<td></td>
<td>char-whitespace?</td>
</tr>
<tr>
<td></td>
<td>char&lt;=?</td>
</tr>
<tr>
<td></td>
<td>char&lt;=?</td>
</tr>
<tr>
<td></td>
<td>char=?</td>
</tr>
<tr>
<td></td>
<td>char=&gt;?</td>
</tr>
<tr>
<td></td>
<td>char&gt;?</td>
</tr>
<tr>
<td></td>
<td>complex?</td>
</tr>
<tr>
<td></td>
<td>cons</td>
</tr>
<tr>
<td></td>
<td>cos</td>
</tr>
<tr>
<td></td>
<td>current-input-port</td>
</tr>
<tr>
<td></td>
<td>current-output-port</td>
</tr>
<tr>
<td>/</td>
<td>delay</td>
</tr>
<tr>
<td></td>
<td>display</td>
</tr>
<tr>
<td></td>
<td>div</td>
</tr>
<tr>
<td></td>
<td>div-and-mod</td>
</tr>
<tr>
<td></td>
<td>div0</td>
</tr>
<tr>
<td></td>
<td>div0-and-mod0</td>
</tr>
<tr>
<td></td>
<td>dynamic-wind</td>
</tr>
<tr>
<td>A</td>
<td>boolean?</td>
</tr>
<tr>
<td></td>
<td>abs</td>
</tr>
<tr>
<td></td>
<td>acos</td>
</tr>
<tr>
<td></td>
<td>angle</td>
</tr>
<tr>
<td></td>
<td>append</td>
</tr>
<tr>
<td></td>
<td>apply</td>
</tr>
<tr>
<td></td>
<td>asin</td>
</tr>
<tr>
<td></td>
<td>assoc</td>
</tr>
<tr>
<td></td>
<td>assq</td>
</tr>
<tr>
<td></td>
<td>assv</td>
</tr>
<tr>
<td></td>
<td>atan</td>
</tr>
<tr>
<td>B</td>
<td>call-with-current-continuation</td>
</tr>
<tr>
<td></td>
<td>call-with-input-file</td>
</tr>
<tr>
<td></td>
<td>call-with-output-file</td>
</tr>
<tr>
<td></td>
<td>call-with-values</td>
</tr>
<tr>
<td></td>
<td>car</td>
</tr>
<tr>
<td></td>
<td>cdr</td>
</tr>
<tr>
<td></td>
<td>ceiling</td>
</tr>
<tr>
<td></td>
<td>ceiling-quotient</td>
</tr>
<tr>
<td></td>
<td>ceiling-remainder</td>
</tr>
<tr>
<td></td>
<td>ceiling/</td>
</tr>
<tr>
<td></td>
<td>centered-quotient</td>
</tr>
<tr>
<td></td>
<td>centered-remainder</td>
</tr>
<tr>
<td></td>
<td>centered/</td>
</tr>
<tr>
<td></td>
<td>char-&gt;integer</td>
</tr>
<tr>
<td></td>
<td>char-alphabetic?</td>
</tr>
<tr>
<td></td>
<td>char-ci&lt;=?</td>
</tr>
<tr>
<td></td>
<td>char-ci&lt;?</td>
</tr>
<tr>
<td></td>
<td>char-ci=?</td>
</tr>
<tr>
<td></td>
<td>char-ci&gt;?</td>
</tr>
<tr>
<td></td>
<td>char-ci&gt;=?</td>
</tr>
<tr>
<td></td>
<td>char-ci?&gt;?</td>
</tr>
<tr>
<td></td>
<td>floor</td>
</tr>
<tr>
<td></td>
<td>floor-quotient</td>
</tr>
<tr>
<td></td>
<td>floor-remainder</td>
</tr>
<tr>
<td></td>
<td>floor/</td>
</tr>
<tr>
<td></td>
<td>for-each</td>
</tr>
<tr>
<td></td>
<td>force</td>
</tr>
</tbody>
</table>
### G

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gcd</td>
<td>116</td>
</tr>
</tbody>
</table>

### I

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imag-part</td>
<td>118</td>
</tr>
<tr>
<td>Inexact-&gt;exact</td>
<td>113</td>
</tr>
<tr>
<td>Inexact?</td>
<td>113</td>
</tr>
<tr>
<td>Input-port?</td>
<td>329</td>
</tr>
<tr>
<td>Integer-&gt;char</td>
<td>133</td>
</tr>
<tr>
<td>Integer?</td>
<td>106</td>
</tr>
<tr>
<td>Interaction-environment</td>
<td>384</td>
</tr>
</tbody>
</table>

### L

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lcm</td>
<td>116</td>
</tr>
<tr>
<td>Length</td>
<td>183</td>
</tr>
<tr>
<td>List</td>
<td>182</td>
</tr>
<tr>
<td>List-&gt;string</td>
<td>144</td>
</tr>
<tr>
<td>List-&gt;symbol</td>
<td>168</td>
</tr>
<tr>
<td>List-&gt;vector</td>
<td>187</td>
</tr>
<tr>
<td>List-ref</td>
<td>183</td>
</tr>
<tr>
<td>List-tail</td>
<td>183</td>
</tr>
<tr>
<td>List?</td>
<td>181</td>
</tr>
<tr>
<td>Load</td>
<td>388</td>
</tr>
<tr>
<td>Log</td>
<td>124</td>
</tr>
</tbody>
</table>

### M

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnitude</td>
<td>118</td>
</tr>
<tr>
<td>Make-polar</td>
<td>118</td>
</tr>
<tr>
<td>Make-rectangular</td>
<td>118</td>
</tr>
<tr>
<td>Make-string</td>
<td>144</td>
</tr>
<tr>
<td>Make-vector</td>
<td>187</td>
</tr>
<tr>
<td>Map</td>
<td>186</td>
</tr>
<tr>
<td>Max</td>
<td>119</td>
</tr>
<tr>
<td>Member</td>
<td>186</td>
</tr>
<tr>
<td>Memq</td>
<td>185</td>
</tr>
<tr>
<td>Memv</td>
<td>186</td>
</tr>
<tr>
<td>Min</td>
<td>119</td>
</tr>
<tr>
<td>Mod</td>
<td>655</td>
</tr>
<tr>
<td>Mod0</td>
<td>655</td>
</tr>
<tr>
<td>Modulo</td>
<td>116</td>
</tr>
</tbody>
</table>

### N

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative?</td>
<td>117</td>
</tr>
<tr>
<td>Newline</td>
<td>349</td>
</tr>
<tr>
<td>Not</td>
<td>104</td>
</tr>
<tr>
<td>Null?</td>
<td>182</td>
</tr>
<tr>
<td>Number-&gt;string</td>
<td>118</td>
</tr>
<tr>
<td>Number?</td>
<td>105</td>
</tr>
</tbody>
</table>

### O

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd?</td>
<td>116</td>
</tr>
<tr>
<td>Open-input-file</td>
<td>342</td>
</tr>
<tr>
<td>Open-output-file</td>
<td>342</td>
</tr>
<tr>
<td>Output-port?</td>
<td>329</td>
</tr>
</tbody>
</table>

### P

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair?</td>
<td>179</td>
</tr>
<tr>
<td>Peek-char</td>
<td>349</td>
</tr>
<tr>
<td>Positive?</td>
<td>117</td>
</tr>
<tr>
<td>Print</td>
<td>383</td>
</tr>
<tr>
<td>Procedure?</td>
<td>258</td>
</tr>
</tbody>
</table>

### Q

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quotient</td>
<td>116</td>
</tr>
</tbody>
</table>

### R

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational?</td>
<td>110</td>
</tr>
<tr>
<td>Read</td>
<td>382</td>
</tr>
<tr>
<td>Read-char</td>
<td>348</td>
</tr>
<tr>
<td>Real-part</td>
<td>118</td>
</tr>
<tr>
<td>Real?</td>
<td>110</td>
</tr>
<tr>
<td>Remainder</td>
<td>116</td>
</tr>
<tr>
<td>Reverse</td>
<td>184</td>
</tr>
<tr>
<td>Round</td>
<td>119</td>
</tr>
<tr>
<td>Round-quotient</td>
<td>119</td>
</tr>
<tr>
<td>Round-remainder</td>
<td>119</td>
</tr>
<tr>
<td>Round/</td>
<td>119</td>
</tr>
</tbody>
</table>

### S

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set-car!</td>
<td>181</td>
</tr>
<tr>
<td>Set-cdr!</td>
<td>181</td>
</tr>
<tr>
<td>Sin</td>
<td>124</td>
</tr>
<tr>
<td>Sqrt</td>
<td>123</td>
</tr>
<tr>
<td>String</td>
<td>144</td>
</tr>
<tr>
<td>String-&gt;list</td>
<td>145</td>
</tr>
<tr>
<td>String-&gt;number</td>
<td>118</td>
</tr>
<tr>
<td>String-&gt;symbol</td>
<td>168</td>
</tr>
<tr>
<td>String-append</td>
<td>155</td>
</tr>
<tr>
<td>String-ci&lt;?</td>
<td>149</td>
</tr>
<tr>
<td>String-ci=?</td>
<td>149</td>
</tr>
<tr>
<td>String-ci&gt;=?</td>
<td>149</td>
</tr>
<tr>
<td>String-ci?&gt;</td>
<td>149</td>
</tr>
<tr>
<td>String-copy</td>
<td>146</td>
</tr>
<tr>
<td>String-fill!</td>
<td>148</td>
</tr>
<tr>
<td>String-length</td>
<td>145</td>
</tr>
<tr>
<td>String-ref</td>
<td>145</td>
</tr>
<tr>
<td>String-set!</td>
<td>147</td>
</tr>
<tr>
<td>String&lt;=?</td>
<td>149</td>
</tr>
<tr>
<td>String&lt;&gt;?</td>
<td>148</td>
</tr>
<tr>
<td>String&gt;=?</td>
<td>148</td>
</tr>
<tr>
<td>String&gt;?</td>
<td>149</td>
</tr>
<tr>
<td>String-&gt;?</td>
<td>149</td>
</tr>
</tbody>
</table>
string? ................................. 143
substring ............................... 146
symbol ................................. 168
symbol->string ........................ 167
symbol-append ......................... 168
symbol? ................................. 167

T

tan ....................................... 124
truncatem ............................... 119
truncate-quotient ..................... 119
truncate-remainder .................... 119
truncate/ ............................... 119

V

values ..................................... 309
vector ................................. 187
vector->list ........................... 187
vector-fill! .............................. 189
vector-length ........................... 188
vector-ref .............................. 188
vector-set! .............................. 188
vector? ................................. 187

W

with-input-from-file ................... 343
with-output-to-file .................... 343
write .................................. 383
write-char ............................. 349

Z

zero? .................................... 117