Guile Reference Manual

Edition 2.0.6, revision 1, for use with Guile 2.0.6

The Guile Developers

This manual documents Guile version 2.0.6.

Copyright (C) 1996, 1997, 2000, 2001, 2002, 2003, 2004, 2005, 2009, 2010, 2011, 2012 Free Software Foundation.

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

3.3.3 Using the Guile REPL ... 25
3.3.4 Summary of Common Syntax ... 25
3.4 The Concept of Closure ... 26
3.4.1 Names, Locations, Values and Environments ... 26
3.4.2 Local Variables and Environments ... 27
3.4.3 Environment Chaining ... 27
3.4.4 Lexical Scope ... 28
3.4.4.1 An Example of Non-Lexical Scoping ... 28
3.4.5 Closure ... 30
3.4.6 Example 1: A Serial Number Generator ... 31
3.4.7 Example 2: A Shared Persistent Variable ... 31
3.4.8 Example 3: The Callback Closure Problem ... 32
3.4.9 Example 4: Object Orientation ... 33
3.5 Further Reading ... 34

4 Programming in Scheme ... 35

4.1 Guile's Implementation of Scheme ... 35
4.2 Invoking Guile ... 35
4.2.1 Command-line Options ... 35
4.2.2 Environment Variables ... 38
4.3 Guile Scripting ... 40
4.3.1 The Top of a Script File ... 40
4.3.2 The Meta Switch ... 40
4.3.3 Command Line Handling ... 42
4.3.4 Scripting Examples ... 43
4.4 Using Guile Interactively ... 45
4.4.1 The Init File, '`~/.guile`' ... 45
4.4.2 Readline ... 46
4.4.3 Value History ... 46
4.4.4 REPL Commands ... 47
4.4.4.1 Help Commands ... 47
4.4.4.2 Module Commands ... 48
4.4.4.3 Language Commands ... 48
4.4.4.4 Compile Commands ... 48
4.4.4.5 Profile Commands ... 49
4.4.4.6 Debug Commands ... 49
4.4.4.7 Inspect Commands ... 50
4.4.4.8 System Commands ... 50
4.4.5 Error Handling ... 51
4.4.6 Interactive Debugging ... 52
4.5 Using Guile in Emacs ... 53
4.6 Using Guile Tools ... 54
4.7 Installing Site Packages ... 54

5 Programming in C ... 57
5.1 Parallel Installations ... 57
5.2 Linking Programs With Guile ... 58
5.2.1 Guile Initialization Functions ... 58
5.2.2 A Sample Guile Main Program ... 59
5.3 Linking Guile with Libraries ... 61
5.3.1 A Sample Guile Extension ... 61
5.4 General concepts for using libguile ... 62
5.4.1 Dynamic Types ... 63
5.4.2 Garbage Collection ... 65
5.4.3 Control Flow ... 66
5.4.4 Asynchronous Signals ... 68
5.4.5 Multi-Threading ... 68
5.5 Defining New Types (Smobs) ... 71
5.5.1 Describing a New Type ... 71
5.5.2 Creating Smob Instances ... 73
5.5.3 Type checking ... 75
5.5.4 Garbage Collecting Smobs ... 76
5.5.5 Garbage Collecting Simple Smobs ... 77
5.5.6 Remembering During Operations ... 78
5.5.7 Double Smobs ... 79
5.5.8 The Complete Example ... 79
5.6 Function Snarfing ... 82
5.7 An Overview of Guile Programming ... 84
5.7.1 How One Might Extend Dia Using Guile ... 84
5.7.1.1 Deciding Why You Want to Add Guile ... 84
5.7.1.2 Four Steps Required to Add Guile ... 85
5.7.1.3 How to Represent Dia Data in Scheme ... 86
5.7.1.4 Writing Guile Primitives for Dia ... 87
5.7.1.5 Providing a Hook for the Evaluation of Scheme Code ... 88
5.7.1.6 Top-level Structure of Guile-enabled Dia ... 88
5.7.1.7 Going Further with Dia and Guile ... 89
5.7.2 Why Scheme is More Hackable Than C ... 91
5.7.3 Example: Using Guile for an Application Testbed ... 91
5.7.4 A Choice of Programming Options ... 92
5.7.4.1 What Functionality is Already Available? ... 92
5.7.4.2 Functional and Performance Constraints ... 93
5.7.4.3 Your Preferred Programming Style ... 93
5.7.4.4 What Controls Program Execution? ... 93
5.7.5 How About Application Users? ... 93
5.8 Autoconf Support ... 95
5.8.1 Autoconf Background ... 95
5.8.2 Autoconf Macros ... 95
5.8.3 Using Autoconf Macros ... 96

6 API Reference ... 99
6.1 Overview of the Guile API ... 99
6.2 Deprecation ... 100
6.3 The SCM Type ... 100
6.4 Initializing Guile ... 101
6.5 Snarfing Macros ... 102
6.6 Simple Generic Data Types ... 104
6.6.1 Booleans ... 104
6.6.2 Numerical data types ... 105
6.6.2.1 Scheme's Numerical "Tower" ... 105
6.6.2.2 Integers ... 106
6.6.2.3 Real and Rational Numbers ... 110
6.6.2.4 Complex Numbers ... 112
6.6.2.5 Exact and Inexact Numbers ... 113
6.6.2.6 Read Syntax for Numerical Data ... 114
6.6.2.7 Operations on Integer Values ... 115
6.6.2.8 Comparison Predicates ... 117
6.6.2.9 Converting Numbers To and From Strings ... 117
6.6.2.10 Complex Number Operations ... 118
6.6.2.11 Arithmetic Functions ... 119
6.6.2.12 Scientific Functions ... 123
6.6.2.13 Bitwise Operations ... 124
6.6.2.14 Random Number Generation ... 127
6.6.3 Characters ... 129
6.6.4 Character Sets ... 133
6.6.4.1 Character Set Predicates/Comparison ... 133
6.6.4.2 Iterating Over Character Sets ... 133
6.6.4.3 Creating Character Sets ... 135
6.6.4.4 Querying Character Sets ... 136
6.6.4.5 Character-Set Algebra ... 137
6.6.4.6 Standard Character Sets ... 138
6.6.5 Strings ... 140
6.6.5.1 String Read Syntax ... 141
6.6.5.2 String Predicates ... 142
6.6.5.3 String Constructors ... 143
6.6.5.4 List/String conversion ... 144
6.6.5.5 String Selection ... 144
6.6.5.6 String Modification ... 146
6.6.5.7 String Comparison ... 147
6.6.5.8 String Searching ... 151
6.6.5.9 Alphabetic Case Mapping ... 153
6.6.5.10 Reversing and Appending Strings ... 154
6.6.5.11 Mapping, Folding, and Unfolding ... 155
6.6.5.12 Miscellaneous String Operations ... 157
6.6.5.13 Conversion to/from C ... 158
6.6.5.14 String Internals ... 161
6.6.6 Bytevectors ... 162
6.6.6.1 Endianness ... 162

6.6.6.2 Manipulating Bytevectors 162
6.6.6.3 Interpreting Bytevector Contents as Integers 164
6.6.6.4 Converting Bytevectors to/from Integer Lists 166
6.6.6.5 Interpreting Bytevector Contents as Floating Point Numbers 167
6.6.6.6 Interpreting Bytevector Contents as Unicode Strings 167
6.6.6.7 Accessing Bytevectors with the Generalized Vector API 168
6.6.6.8 Accessing Bytevectors with the SRFI-4 API 168
6.6.7 Symbols 168
6.6.7.1 Symbols as Discrete Data 169
6.6.7.2 Symbols as Lookup Keys 170
6.6.7.3 Symbols as Denoting Variables 171
6.6.7.4 Operations Related to Symbols 172
6.6.7.5 Function Slots and Property Lists 175
6.6.7.6 Extended Read Syntax for Symbols 176
6.6.7.7 Uninterned Symbols 176
6.6.8 Keywords 178
6.6.8.1 Why Use Keywords? 178
6.6.8.2 Coding With Keywords 179
6.6.8.3 Keyword Read Syntax 180
6.6.8.4 Keyword Procedures 181
6.6.9 “Functionality-Centric” Data Types 181
6.7 Compound Data Types 182
6.7.1 Pairs 182
6.7.2 Lists 185
6.7.2.1 List Read Syntax 185
6.7.2.2 List Predicates 185
6.7.2.3 List Constructors 186
6.7.2.4 List Selection 186
6.7.2.5 Append and Reverse 187
6.7.2.6 List Modification 188
6.7.2.7 List Searching 189
6.7.2.8 List Mapping 190
6.7.3 Vectors 190
6.7.3.1 Read Syntax for Vectors 190
6.7.3.2 Dynamic Vector Creation and Validation 191
6.7.3.3 Accessing and Modifying Vector Contents 191
6.7.3.4 Vector Accessing from C 193
6.7.3.5 Uniform Numeric Vectors 194
6.7.4 Bit Vectors 194
6.7.5 Generalized Vectors 197
6.7.6 Arrays 198
6.7.6.1 Array Syntax 198
6.7.6.2 Array Procedures 199
6.7.6.3 Shared Arrays 203
6.7.6.4 Accessing Arrays from C 205

6.7.7 VLists . . . 210
6.7.8 Records . . . 212
6.7.9 Structures . . . 214
6.7.9.1 Vtables . . . 214
6.7.9.2 Structure Basics . . . 215
6.7.9.3 Vtable Contents . . . 216
6.7.9.4 Vtable Vtables . . . 217
6.7.10 Dictionary Types . . . 219
6.7.11 Association Lists . . . 220
6.7.11.1 Alist Key Equality . . . 220
6.7.11.2 Adding or Setting Alist Entries . . . 220
6.7.11.3 Retrieving Alist Entries . . . 222
6.7.11.4 Removing Alist Entries . . . 223
6.7.11.5 Sloppy Alist Functions . . . 224
6.7.11.6 Alist Example . . . 225
6.7.12 VList-Based Hash Lists or “VHashes” . . . 226
6.7.13 Hash Tables . . . 228
6.7.13.1 Hash Table Examples . . . 228
6.7.13.2 Hash Table Reference . . . 230
6.8 Smobs . . . 233
6.9 Procedures . . . 236
6.9.1 Lambda: Basic Procedure Creation . . . 236
6.9.2 Primitive Procedures . . . 237
6.9.3 Compiled Procedures . . . 238
6.9.4 Optional Arguments . . . 240
6.9.4.1 lambda* and define* . . . 240
6.9.4.2 (ice-9 optargs) . . . 242
6.9.5 Case-lambda . . . 243
6.9.6 Higher-Order Functions . . . 244
6.9.7 Procedure Properties and Meta-information . . . 245
6.9.8 Procedures with Setters . . . 246
6.9.9 Inlinable Procedures . . . 247
6.10 Macros . . . 248
6.10.1 Defining Macros . . . 248
6.10.2 Syntax-rules Macros . . . 249
6.10.2.1 Patterns . . . 250
6.10.2.2 Hygiene . . . 252
6.10.2.3 Shorthands . . . 253
6.10.2.4 Further Information . . . 253
6.10.3 Support for the `syntax-case` System . . . 253
6.10.3.1 Why `syntax-case`? . . . 255
6.10.4 Syntax Transformer Helpers . . . 258
6.10.5 Lisp-style Macro Definitions . . . 260
6.10.6 Identifier Macros . . . 261
6.10.7 Syntax Parameters . . . 262
6.10.8 Eval-when . . . 263
6.10.9 Internal Macros . . . 264
6.11 General Utility Functions . . . 265

6.11.1 Equality ... 265
6.11.2 Object Properties ... 267
6.11.3 Sorting ... 268
6.11.4 Copying Deep Structures ... 270
6.11.5 General String Conversion ... 270
6.11.6 Hooks ... 270
6.11.6.1 Hook Usage by Example ... 270
6.11.6.2 Hook Reference ... 271
6.11.6.3 Handling Scheme-level hooks from C code ... 273
6.11.6.4 Hooks For C Code ... 273
6.11.6.5 Hooks for Garbage Collection ... 274
6.11.6.6 Hooks into the Guile REPL ... 275
6.12 Definitions and Variable Bindings ... 275
6.12.1 Top Level Variable Definitions ... 275
6.12.2 Local Variable Bindings ... 277
6.12.3 Internal definitions ... 278
6.12.4 Querying variable bindings ... 279
6.13 Controlling the Flow of Program Execution ... 279
6.13.1 Sequencing and Splicing ... 280
6.13.2 Simple Conditional Evaluation ... 281
6.13.3 Conditional Evaluation of a Sequence of Expressions ... 282
6.13.4 Iteration mechanisms ... 283
6.13.5 Prompts ... 285
6.13.5.1 Prompt Primitives ... 285
6.13.5.2 Shift, Reset, and All That ... 287
6.13.6 Continuations ... 288
6.13.7 Returning and Accepting Multiple Values ... 290
6.13.8 Exceptions ... 291
6.13.8.1 Exception Terminology ... 292
6.13.8.2 Catching Exceptions ... 292
6.13.8.3 Throw Handlers ... 294
6.13.8.4 Throwing Exceptions ... 296
6.13.8.5 How Guile Implements Exceptions ... 297
6.13.9 Procedures for Signaling Errors ... 297
6.13.10 Dynamic Wind ... 298
6.13.11 How to Handle Errors ... 301
6.13.11.1 C Support ... 302
6.13.11.2 Signalling Type Errors ... 303
6.13.12 Continuation Barriers ... 303
6.14 Input and Output ... 304
6.14.1 Ports ... 304
6.14.2 Reading ... 306
6.14.3 Writing ... 308
6.14.4 Closing ... 309
6.14.5 Random Access ... 310
6.14.6 Line Oriented and Delimited Text ... 310
6.14.7 Block reading and writing ... 312
6.14.8 Default Ports for Input, Output and Errors ... 313

6.14.9 Types of Port ... 314
6.14.9.1 File Ports ... 314
6.14.9.2 String Ports ... 317
6.14.9.3 Soft Ports ... 318
6.14.9.4 Void Ports ... 319
6.14.10 R6RS I/O Ports ... 319
6.14.10.1 File Names ... 319
6.14.10.2 File Options ... 320
6.14.10.3 Buffer Modes ... 320
6.14.10.4 Transcoders ... 321
6.14.10.5 The End-of-File Object ... 324
6.14.10.6 Port Manipulation ... 324
6.14.10.7 Input Ports ... 325
6.14.10.8 Binary Input ... 326
6.14.10.9 Textual Input ... 328
6.14.10.10 Output Ports ... 330
6.14.10.11 Binary Output ... 331
6.14.10.12 Textual Output ... 332
6.14.11 Using and Extending Ports in C ... 332
6.14.11.1 C Port Interface ... 332
6.14.11.2 Port Implementation ... 334
6.15 Regular Expressions ... 335
6.15.1 Regexp Functions ... 336
6.15.2 Match Structures ... 340
6.15.3 Backslash Escapes ... 341
6.16 LALR(1) Parsing ... 342
6.17 Reading and Evaluating Scheme Code ... 343
6.17.1 Scheme Syntax: Standard and Guile Extensions ... 343
6.17.1.1 Expression Syntax ... 343
6.17.1.2 Comments ... 345
6.17.1.3 Block Comments ... 345
6.17.1.4 Case Sensitivity ... 345
6.17.1.5 Keyword Syntax ... 346
6.17.1.6 Reader Extensions ... 346
6.17.2 Reading Scheme Code ... 346
6.17.3 Writing Scheme Values ... 347
6.17.4 Procedures for On the Fly Evaluation ... 348
6.17.5 Compiling Scheme Code ... 350
6.17.6 Loading Scheme Code from File ... 352
6.17.7 Load Paths ... 353
6.17.8 Character Encoding of Source Files ... 355
6.17.9 Delayed Evaluation ... 356
6.17.10 Local Evaluation ... 357
6.17.11 Local Inclusion ... 357
6.18 Memory Management and Garbage Collection ... 358
6.18.1 Function related to Garbage Collection ... 358
6.18.2 Memory Blocks ... 359
6.18.2.1 Upgrading from scm_must_malloc et al. ... 361

6.18.3 Weak References ... 362
6.18.3.1 Weak hash tables ... 363
6.18.3.2 Weak vectors ... 363
6.18.4 Guardians ... 364
6.19 Modules ... 364
6.19.1 General Information about Modules ... 365
6.19.2 Using Guile Modules ... 365
6.19.3 Creating Guile Modules ... 367
6.19.4 Modules and the File System ... 370
6.19.5 R6RS Version References ... 371
6.19.6 R6RS Libraries ... 372
6.19.7 Variables ... 373
6.19.8 Module System Reflection ... 375
6.19.9 Accessing Modules from C ... 376
6.19.10 Included Guile Modules ... 379
6.19.11 provide and require ... 381
6.19.12 Environments ... 381
6.20 Foreign Function Interface ... 382
6.20.1 Foreign Libraries ... 382
6.20.2 Foreign Functions ... 384
6.20.3 C Extensions ... 385
6.20.4 Modules and Extensions ... 386
6.20.5 Foreign Pointers ... 388
6.20.5.1 Foreign Types ... 388
6.20.5.2 Foreign Variables ... 389
6.20.5.3 Void Pointers and Byte Access ... 390
6.20.5.4 Foreign Structs ... 392
6.20.6 Dynamic FFI ... 393
6.21 Threads, Mutexes, Asyncs and Dynamic Roots ... 396
6.21.1 Arbiters ... 396
6.21.2 Asyncs ... 397
6.21.2.1 System asyncs ... 397
6.21.2.2 User asyncs ... 398
6.21.3 Threads ... 398
6.21.4 Mutexes and Condition Variables ... 400
6.21.5 Blocking in Guile Mode ... 404
6.21.6 Critical Sections ... 405
6.21.7 Fluids and Dynamic States ... 405
6.21.8 Parameters ... 408
6.21.9 Futures ... 409
6.21.10 Parallel forms ... 410
6.22 Configuration, Features and Runtime Options ... 412
6.22.1 Configuration, Build and Installation ... 412
6.22.2 Feature Tracking ... 414
6.22.2.1 Feature Manipulation ... 414
6.22.2.2 Common Feature Symbols ... 414
6.22.3 Runtime Options ... 416
6.22.3.1 Examples of option use ... 416

6.23 Support for Other Languages ... 417
6.23.1 Using Other Languages ... 417
6.23.2 Emacs Lisp ... 418
6.23.2.1 Nil ... 418
6.23.2.2 Equality ... 418
6.23.2.3 Dynamic Binding ... 420
6.23.2.4 Other Elisp Features ... 420
6.23.3 ECMAScript ... 420
6.24 Support for Internationalization ... 420
6.24.1 Internationalization with Guile ... 421
6.24.2 Text Collation ... 422
6.24.3 Character Case Mapping ... 423
6.24.4 Number Input and Output ... 423
6.24.5 Accessing Locale Information ... 424
6.24.6 Gettext Support ... 427
6.25 Debugging Infrastructure ... 429
6.25.1 Evaluation and the Scheme Stack ... 429
6.25.1.1 Stack Capture ... 430
6.25.1.2 Stacks ... 431
6.25.1.3 Frames ... 431
6.25.2 Source Properties ... 432
6.25.3 Programmatic Error Handling ... 434
6.25.3.1 Catching Exceptions ... 434
6.25.3.2 Capturing the full error stack ... 435
6.25.3.3 Pre-Unwind Debugging ... 437
6.25.3.4 Debug options ... 438
6.25.4 Traps ... 440
6.25.4.1 VM Hooks ... 440
6.25.4.2 Trap Interface ... 442
6.25.4.3 Low-Level Traps ... 442
6.25.4.4 Tracing Traps ... 444
6.25.4.5 Trap States ... 446
6.25.4.6 High-Level Traps ... 447
6.26 Code Coverage Reports ... 448

7 Guile Modules ... 451

7.1 SLIB ... 451
7.1.1 SLIB installation ... 451
7.1.2 JACAL ... 452
7.2 POSIX System Calls and Networking ... 452
7.2.1 POSIX Interface Conventions ... 452
7.2.2 Ports and File Descriptors ... 453
7.2.3 File System ... 460
7.2.4 User Information ... 466
7.2.5 Time ... 468
7.2.6 Runtime Environment ... 471
7.2.7 Processes ... 473
7.2.8 Signals ... 479

7.2.9 Terminals and Ptys ... 482
7.2.10 Pipes ... 482
7.2.11 Networking ... 484
7.2.11.1 Network Address Conversion ... 484
7.2.11.2 Network Databases ... 485
7.2.11.3 Network Socket Address ... 492
7.2.11.4 Network Sockets and Communication ... 493
7.2.11.5 Network Socket Examples ... 499
7.2.12 System Identification ... 500
7.2.13 Locales ... 501
7.2.14 Encryption ... 501
7.3 HTTP, the Web, and All That ... 502
7.3.1 Types and the Web ... 502
7.3.2 Universal Resource Identifiers ... 504
7.3.3 The Hyper-Text Transfer Protocol ... 506
7.3.4 HTTP Headers ... 509
7.3.4.1 HTTP Header Types ... 509
7.3.4.2 General Headers ... 510
7.3.4.3 Entity Headers ... 511
7.3.4.4 Request Headers ... 512
7.3.4.5 Response Headers ... 515
7.3.5 Transfer Codings ... 516
7.3.6 HTTP Requests ... 517
7.3.6.1 An Important Note on Character Sets ... 517
7.3.6.2 Request API ... 517
7.3.7 HTTP Responses ... 519
7.3.8 Web Client ... 521
7.3.9 Web Server ... 522
7.3.10 Web Examples ... 525
7.3.10.1 Hello, World! ... 525
7.3.10.2 Inspecting the Request ... 525
7.3.10.3 Higher-Level Interfaces ... 526
7.3.10.4 Conclusion ... 528
7.4 The (ice-9 getopt-long) Module ... 528
7.4.1 A Short getopt-long Example ... 528
7.4.2 How to Write an Option Specification ... 529
7.4.3 Expected Command Line Format ... 530
7.4.4 Reference Documentation for `getopt-long` ... 531
7.4.5 Reference Documentation for `option-ref` ... 532
7.5 SRFI Support Modules ... 533
7.5.1 About SRFI Usage ... 533
7.5.2 SRFI-0 - cond-expand ... 533
7.5.3 SRFI-1 - List library ... 535
7.5.3.1 Constructors ... 535
7.5.3.2 Predicates ... 535
7.5.3.3 Selectors ... 536
7.5.3.4 Length, Append, Concatenate, etc. ... 537
7.5.3.5 Fold, Unfold & Map ... 538

7.5.3.6 Filtering and Partitioning ... 541
7.5.3.7 Searching ... 542
7.5.3.8 Deleting ... 544
7.5.3.9 Association Lists ... 544
7.5.3.10 Set Operations on Lists ... 545
7.5.4 SRFI-2 - and-let* ... 548
7.5.5 SRFI-4 - Homogeneous numeric vector datatypes ... 549
7.5.5.1 SRFI-4 - Overview ... 549
7.5.5.2 SRFI-4 - API ... 550
7.5.5.3 SRFI-4 - Generic operations ... 556
7.5.5.4 SRFI-4 - Relation to bytevectors ... 557
7.5.5.5 SRFI-4 - Guile extensions ... 557
7.5.6 SRFI-6 - Basic String Ports ... 558
7.5.7 SRFI-8 - receive ... 558
7.5.8 SRFI-9 - define-record-type ... 558
Non-toplevel Record Definitions ... 559
Custom Printers ... 560
7.5.9 SRFI-10 - Hash-Comma Reader Extension ... 560
7.5.10 SRFI-11 - let-values ... 562
7.5.11 SRFI-13 - String Library ... 562
7.5.12 SRFI-14 - Character-set Library ... 562
7.5.13 SRFI-16 - case-lambda ... 562
7.5.14 SRFI-17 - Generalized set! ... 562
7.5.15 SRFI-18 - Multithreading support ... 563
7.5.15.1 SRFI-18 Threads ... 563
7.5.15.2 SRFI-18 Mutexes ... 565
7.5.15.3 SRFI-18 Condition variables ... 566
7.5.15.4 SRFI-18 Time ... 566
7.5.15.5 SRFI-18 Exceptions ... 567
7.5.16 SRFI-19 - Time/Date Library ... 568
7.5.16.1 SRFI-19 Introduction ... 568
7.5.16.2 SRFI-19 Time ... 568
7.5.16.3 SRFI-19 Date ... 570
7.5.16.4 SRFI-19 Time/Date conversions ... 571
7.5.16.5 SRFI-19 Date to string ... 572
7.5.16.6 SRFI-19 String to date ... 574
7.5.17 SRFI-23 - Error Reporting ... 575
7.5.18 SRFI-26 - specializing parameters ... 575
7.5.19 SRFI-27 - Sources of Random Bits ... 576
7.5.19.1 The Default Random Source ... 576
7.5.19.2 Random Sources ... 577
7.5.19.3 Obtaining random number generator procedures ... 577
7.5.20 SRFI-30 - Nested Multi-line Comments ... 578
7.5.21 SRFI-31 - A special form 'rec' for recursive evaluation ... 578
7.5.22 SRFI-34 - Exception handling for programs ... 578
7.5.23 SRFI-35 - Conditions ... 578
7.5.24 SRFI-37 - args-fold ... 581

7.5.25 SRFI-38 - External Representation for Data With Shared Structure ... 582
7.5.26 SRFI-39 - Parameters ... 583
7.5.27 SRFI-42 - Eager Comprehensions ... 584
7.5.28 SRFI-45 - Primitives for Expressing Iterative Lazy Algorithms ... 584
7.5.29 SRFI-55 - Requiring Features ... 585
7.5.30 SRFI-60 - Integers as Bits ... 586
7.5.31 SRFI-61 - A more general `cond` clause ... 587
7.5.32 SRFI-67 - Compare procedures ... 587
7.5.33 SRFI-69 - Basic hash tables ... 588
7.5.33.1 Creating hash tables ... 588
7.5.33.2 Accessing table items ... 589
7.5.33.3 Table properties ... 589
7.5.33.4 Hash table algorithms ... 590
7.5.34 SRFI-88 Keyword Objects ... 590
7.5.35 SRFI-98 Accessing environment variables ... 591
7.6 R6RS Support ... 591
7.6.1 Incompatibilities with the R6RS ... 591
7.6.2 R6RS Standard Libraries ... 592
7.6.2.1 Library Usage ... 592
7.6.2.2 rnrs base ... 593
7.6.2.3 rnrs unicode ... 600
7.6.2.4 rnrs bytevectors ... 601
7.6.2.5 rnrs lists ... 601
7.6.2.6 rnrs sorting ... 602
7.6.2.7 rnrs control ... 603
7.6.2.8 R6RS Records ... 603
7.6.2.9 rnrs records syntactic ... 604
7.6.2.10 rnrs records procedural ... 605
7.6.2.11 rnrs records inspection ... 606
7.6.2.12 rnrs exceptions ... 607
7.6.2.13 rnrs conditions ... 608
7.6.2.14 I/O Conditions ... 611
7.6.2.15 rnrs io ports ... 612
7.6.2.16 rnrs io simple ... 612
7.6.2.17 rnrs files ... 613
7.6.2.18 rnrs programs ... 613
7.6.2.19 rnrs arithmetic fixnums ... 614
7.6.2.20 rnrs arithmetic flonums ... 616
7.6.2.21 rnrs arithmetic bitwise ... 619
7.6.2.22 rnrs syntax-case ... 620
7.6.2.23 rnrs hashtables ... 621
7.6.2.24 rnrs enums ... 623
7.6.2.25 rnrs ... 624
7.6.2.26 rnrs eval ... 624
7.6.2.27 rnrs mutable-pairs ... 625
7.6.2.28 rnrs mutable-strings ... 625

7.6.2.29 rnrs r5rs ... 625
7.7 Pattern Matching ... 626
7.8 Readline Support ... 628
7.8.1 Loading Readline Support ... 628
7.8.2 Readline Options ... 629
7.8.3 Readline Functions ... 630
7.8.3.1 Readline Port ... 630
7.8.3.2 Completion ... 631
7.9 Pretty Printing ... 632
7.10 Formatted Output ... 633
7.11 File Tree Walk ... 644
7.12 Queues ... 649
7.13 Streams ... 650
7.14 Buffered Input ... 652
7.15 Expect ... 653
7.16 `sxml-match`: Pattern Matching of SXML ... 655
Syntax ... 656
Matching XML Elements ... 657
Ellipses in Patterns ... 658
Ellipses in Quasiquote'd Output ... 658
Matching Nodesets ... 658
Matching the "Rest" of a Nodeset ... 658
Matching the Unmatched Attributes ... 659
Default Values in Attribute Patterns ... 659
Guards in Patterns ... 659
Catamorphisms ... 659
Named-Catamorphisms ... 660
`sxml-match-let` and `sxml-match-let*` ... 661
7.17 The Scheme shell (scsh) ... 661

8 Standard Library ... 663

8.1 (statprof) ... 663
8.1.1 Overview ... 663
8.1.2 Implementation notes ... 664
8.1.3 Usage ... 664
8.2 (sxml apply-templates) ... 666
8.2.1 Overview ... 666
8.2.2 Usage ... 667
8.3 (sxml fold) ... 667
8.3.1 Overview ... 667
8.3.2 Usage ... 667
8.4 (sxml simple) ... 669
8.4.1 Overview ... 669
8.4.2 Usage ... 669
8.5 (sxml ssax) ... 669
8.5.1 Overview ... 669
8.5.2 Usage ... 671
8.6 (sxml ssax input-parse) ... 672

8.6.1 Overview ... 672
8.6.2 Usage ... 672
8.7 (sxml transform) ... 673
8.7.1 Overview ... 673
8.7.2 Usage ... 673
8.8 (sxml xpath) ... 674
8.8.1 Overview ... 674
8.8.2 Usage ... 675
8.9 (texinfo) ... 676
8.9.1 Overview ... 676
8.9.2 Usage ... 677
8.10 (texinfo docbook) ... 677
8.10.1 Overview ... 677
8.10.2 Usage ... 677
8.11 (texinfo html) ... 678
8.11.1 Overview ... 678
8.11.2 Usage ... 679
8.12 (texinfo indexing) ... 679
8.12.1 Overview ... 679
8.12.2 Usage ... 679
8.13 (texinfo string-utils) ... 679
8.13.1 Overview ... 679
8.13.2 Usage ... 679
8.14 (texinfo plain-text) ... 682
8.14.1 Overview ... 682
8.14.2 Usage ... 682
8.15 (texinfo serialize) ... 682
8.15.1 Overview ... 682
8.15.2 Usage ... 683
8.16 (texinfo reflection) ... 683
8.16.1 Overview ... 683
8.16.2 Usage ... 683

9 GOOPS ... 685

9.1 Copyright Notice ... 685
9.2 Class Definition ... 685
9.3 Instance Creation and Slot Access ... 686
9.4 Slot Options ... 687
9.5 Illustrating Slot Description ... 690
9.6 Methods and Generic Functions ... 692
9.6.1 Accessors ... 694
9.6.2 Extending Primitives ... 694
9.6.3 Merging Generics ... 694
9.6.4 Next-method ... 695
9.6.5 Generic Function and Method Examples ... 696
9.6.6 Handling Invocation Errors ... 699
9.7 Inheritance ... 699
9.7.1 Class Precedence List ... 700

9.7.2 Sorting Methods ... 701
9.8 Introspection ... 702
9.8.1 Classes ... 702
9.8.2 Instances ... 703
9.8.3 Slots ... 703
9.8.4 Generic Functions ... 705
9.8.5 Accessing Slots ... 706
9.9 Error Handling ... 708
9.10 GOOPS Object Miscellany ... 708
9.11 The Metaobject Protocol ... 709
9.11.1 Metaobjects and the Metaobject Protocol ... 709
9.11.2 Metaclasses ... 711
9.11.3 MOP Specification ... 712
9.11.4 Instance Creation Protocol ... 712
9.11.5 Class Definition Protocol ... 713
9.11.6 Customizing Class Definition ... 716
9.11.7 Method Definition ... 718
9.11.8 Method Definition Internals ... 718
9.11.9 Generic Function Internals ... 719
9.11.10 Generic Function Invocation ... 720
9.12 Redefining a Class ... 721
9.12.1 Default Class Redefinition Behaviour ... 721
9.12.2 Customizing Class Redefinition ... 721
9.13 Changing the Class of an Instance ... 722

10 Guile Implementation ... 725

10.1 A Brief History of Guile ... 725
10.1.1 The Emacs Thesis ... 725
10.1.2 Early Days ... 725
10.1.3 A Scheme of Many Maintainers ... 726
10.1.4 A Timeline of Selected Guile Releases ... 727
10.1.5 Status, or: Your Help Needed ... 728
10.2 Data Representation ... 729
10.2.1 A Simple Representation ... 730
10.2.2 Faster Integers ... 730
10.2.3 Cheaper Pairs ... 732
10.2.4 Conservative Garbage Collection ... 733
10.2.5 The SCM Type in Guile ... 734
10.2.5.1 Relationship between `SCM` and `scm_t_bits` ... 734
10.2.5.2 Immediate objects ... 735
10.2.5.3 Non-immediate objects ... 736
10.2.5.4 Allocating Cells ... 736
10.2.5.5 Heap Cell Type Information ... 737
10.2.5.6 Accessing Cell Entries ... 737
10.3 A Virtual Machine for Guile ... 738
10.3.1 Why a VM? ... 738
10.3.2 VM Concepts ... 739
10.3.3 Stack Layout ... 740

10.3.4 Variables and the VM 741
10.3.5 Compiled Procedures are VM Programs 742
10.3.6 Instruction Set 743
10.3.6.1 Lexical Environment Instructions 743
10.3.6.2 Top-Level Environment Instructions 745
10.3.6.3 Procedure Call and Return Instructions 746
10.3.6.4 Function Prologue Instructions 748
10.3.6.5 Trampoline Instructions 749
10.3.6.6 Branch Instructions 750
10.3.6.7 Data Constructor Instructions 751
10.3.6.8 Loading Instructions 752
10.3.6.9 Dynamic Environment Instructions 753
10.3.6.10 Miscellaneous Instructions 754
10.3.6.11 Inlined Scheme Instructions 755
10.3.6.12 Inlined Mathematical Instructions 756
10.3.6.13 Inlined Bytevector Instructions 756
10.4 Compiling to the Virtual Machine 757
10.4.1 Compiler Tower 758
10.4.2 The Scheme Compiler 759
10.4.3 Tree-IL 761
10.4.4 GLIL 765
10.4.5 Assembly 768
10.4.6 Bytecode and Objcode 769
10.4.7 Writing New High-Level Languages 770
10.4.8 Extending the Compiler 770

Appendix A GNU Free Documentation License 773

Concept Index 781

Procedure Index 787

Variable Index 827

Type Index 831

R5RS Index 833

Preface

This manual describes how to use Guile, GNU's Ubiquitous Intelligent Language for Extensions. It relates particularly to Guile version 2.0.6.

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 773).

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^n 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^5 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 591.

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 533.

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 `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 2.0.6, so the file you should grab is:

`ftp://ftp.gnu.org/gnu/guile/guile-2.0.6.tar.gz`

To unbundle Guile use the instruction

```
zcat guile-2.0.6.tar.gz | tar xvf -
```

which will create a directory called '`guile-2.0.6`' with all the sources. You can look at the file '`INSTALL`' for detailed instructions on how to build and install Guile, but you should be able to just do

```
cd guile-2.0.6
./configure
make
make install
```

This will install the Guile executable '`guile`', the Guile library '`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[5] Report on the Algorithmic Language Scheme", we have included the report in the Guile distribution; see Section "Introduction" in *Revised(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

We use some conventions in this manual.

- For some procedures, notably type predicates, we use "iff" to mean "if and only if". The construct is usually something like: 'Return *val* iff *condition*', where *val* is usually "`#t`" or "non-`#f`". This typically means that *val* is returned if *condition* holds, and that '`#f`' is returned otherwise. To clarify: *val* will **only** be returned when *condition* is true.
- 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:

```
(+ 1 2)
⇒ 3
```

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

```
(begin (display 1) (newline) 'hooray)
⊣ 1
⇒ hooray
```

As you can see, this code prints '`1`' (denoted by '⊣'), and returns `hooray` (denoted by '⇒').

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 40, 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-2.0`
```

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_make_real (j0 (scm_num2dbl (x, "j0")));
}

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-2.0` \
  -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 54.

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 6.19 [Modules], page 364.

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
scheme@(guile-user)> (use-modules (ice-9 popen))
scheme@(guile-user)> (use-modules (ice-9 rdelim))
scheme@(guile-user)> (define p (open-input-pipe "ls -l"))
scheme@(guile-user)> (read-line p)
$1 = "total 30"
scheme@(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 54.

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/2.0/extensions/libguile-bessel.so
... ELF 32-bit LSB shared object ...
$ guile
scheme@(guile-user)> (use-modules (math bessel))
scheme@(guile-user)> (j0 2)
$1 = 0.223890779141236
```

See Section 6.20.4 [Modules and Extensions], page 386, 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 – `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 `define` syntax like this:

```
(define variable-name value)
```

This makes a new variable called *variable-name* and stores *value* in it as the variable's initial value. For example:

```
;; Make a variable ‘x’ with initial numeric value 1.
(define x 1)

;; Make a variable ‘organization’ with an initial string value.
(define organization "Free Software Foundation")
```

(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 `define` is replaced by the Scheme syntax `set!`, like this:

```
(set! variable-name new-value)
```

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

```
;; Change the value of ‘x’ to 5.
(set! x 5)

;; Change the value of ‘organization’ to the FSF’s street number.
(set! organization 545)
```

In these examples, *value* and *new-value* are literal numeric or string values. In general, however, *value* and *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 `set!` example does...

```
(set! x (+ x 1))
```

(Note: this is not a complete description of `define` and `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 `define` syntax that can be used when defining new procedures.
- Section 6.9.8 [Procedures with Setters], page 246, to read about an alternative form of the `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 278, to read about using `define` other than at top level in a Scheme program, including a discussion of when it works to use `define` rather than `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 `lambda` is used to create new procedures, and conclude by presenting the various shorthand forms of `define` that can be used instead of writing an explicit `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 `open-input-file`, what we actually mean is that there is a pre-defined top level variable called `open-input-file`, whose value is a procedure that implements what R5RS says that `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 *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, `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 `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 `lambda` syntax.

For example, the value of the following Scheme expression

```
(lambda (name address) expression ...)
```

is a newly created procedure that takes two arguments: `name` and `address`. The behaviour of the new procedure is determined by the sequence of *expression*s in the *body* of the procedure definition. (Typically, these *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 *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 lambda expression might look like this:

```
(lambda (name address)
  (string-append "Name=" name ":Address=" address))
```

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

```
((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 —

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

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

```
(make-combined-string "FSF" "Cambridge")
```

Which has exactly the same result.

It's important to note that procedures created using `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 236.

(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 some 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[1] 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"`
- 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.

[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*.

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:

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

In the outermost expression, *procedure* is `string-length` and *arg1* is `(string-append "/home" "/" "andrew")`.

- Evaluation of `string-length`, which is a variable, gives a procedure value that implements the expected behaviour for "string-length".
- Evaluation of `(string-append "/home" "/" "andrew")`, which is another procedure invocation expression, means evaluating each of
  - `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 *procedure* with the value obtained from *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 *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.

```
(if (string=? (read-answer "Should I delete this file?")
              "yes")
    (delete-file file))
```

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

Therefore `if` must be special syntax, not a procedure. Other special syntaxes that we have already met are `define`, `set!` and `lambda`. `define` and `set!` are syntax because they need to know the variable *name* that is given as the first argument in a `define` or `set!` expression, not that variable's value. `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,

```
(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 `last` function (see Section 7.5.3.3 [SRFI-1 Selectors], page 536) returning the last element of a list,

```
(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, `my-last` applies itself to the `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,

- `and` — last expression
- `begin` — last expression
- `case` — last expression in each clause
- `cond` — last expression in each clause, and the call to a `=>` procedure is a tail call
- `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 542).

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 236) is used to construct procedure objects.

`define` (see Section 6.12.1 [Top Level], page 275) is used to create a new variable and set its initial value.

`set!` (see Section 6.12.1 [Top Level], page 275) is used to modify an existing variable's value.

`let`, `let*` and `letrec` (see Section 6.12.2 [Local Bindings], page 277) 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 280) 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 281) 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 281) 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 282) 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 282) 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 277). 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.

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

(define area
  (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 `s` created by the expression:

```
(define s "Some beans, my lord!")
```

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

Within the `let` body, the binding for `s` in the local environment is said to *shadow* the binding for `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.

```
(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)))
```

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 `french-currency-string` so that it mostly works without changing `currency-string`, but the fix is inelegant, and susceptible to interrupts that could leave the `currency-abbreviation` variable in the wrong state:

```
(define (french-currency-string units hundredths)
  (set! currency-abbreviation "FRF")
  (let ((result (currency-string units hundredths)))
    (set! currency-abbreviation "USD")
    result))
```

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

3.4.5 Closure

Consider a `let` expression that doesn't contain any `lambda`s:

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

When the Scheme interpreter evaluates this, it

- creates a new environment with a reference to the environment that was current when it encountered the `let`
- creates a variable binding for `s` in the new environment, with value given by `(/ (+ a b c) 2)`
- evaluates the expression in the body of the `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 `let`, using the value `V` as the value of the `let` expression, in the context of the containing environment.

After the `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 `let` body contains a `lambda` expression, however, the local environment is *not* forgotten. Instead, it becomes associated with the procedure that is created by the `lambda` expression, and is reinstated every time that that procedure is called. In detail, this works as follows.

- When the Scheme interpreter evaluates a `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 `define`ing 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.

```
(my-account 'get-balance)
⇒
0

(my-account 'withdraw 5)
⇒
-5

(my-account 'deposit 396)
⇒
391

(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^n 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 43.

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.

`-x extension`
Add *extension* to the front of Guile's load extension list (see Section 6.17.7 [Load Paths], page 353). 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*.

For compatibility with some versions of Guile 1.4, you can also use the form `(symbol ...)` (that is, a list of only symbols that doesn't start with `@`), which is equivalent to `(@ (symbol ...) main)`, or `(symbol ...) symbol` (that is, a list of only symbols followed by a symbol), which is equivalent to `(@ (symbol ...) symbol)`. We recommend to use the equivalent forms directly since they correspond to the `(@ ...)` read syntax that can be used in normal code. See Section 6.19.2 [Using Guile Modules], page 365 and Section 4.3.4 [Scripting Examples], page 43.

`-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 43.

`\` Read more command-line arguments, starting from the second line of the script file. See Section 4.3.2 [The Meta Switch], page 40.

`--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 6.25.4.1 [VM Hooks], page 440, 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 45.

`--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 a path to 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 53 for more details.

Note that opening a port allows anyone who can connect to that port—in the TCP case, any local user—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.

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.

`-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 471.

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 6.17.5 [Compilation], page 350.

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_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, which will be prefixed to the value of the default search path stored in %load-compiled-path.

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/2.0/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, which will be prefixed to the value of the default search path stored in %load-path.

Here is an example using the Bash shell that adds the current directory and the parent of the current directory to %load-path:

```
$ env GUILE_LOAD_PATH=".:.." \
guile -c '(display %load-path) (newline)'
(. .. /usr/local/share/guile/2.0 \
/usr/local/share/guile/site/2.0 \
/usr/local/share/guile/site /usr/local/share/guile)
```

(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 elimination. GUILE_WARN_DEPRECATED 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'.

`LTDL_LIBRARY_PATH`

Guile now adds its install prefix to the `LTDL_LIBRARY_PATH`.

Users may now install Guile in non-standard directories and run '/path/to/bin/guile', without having also to set `LTDL_LIBRARY_PATH` to include '/path/to/lib'.

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 40.
- 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 6.17.8 [Character Encoding of Source Files], page 355.
- 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

```
#! /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

```
("./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:

```
#! /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 40.)

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 528.

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 -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:

```
#!/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:

```
#!/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:

```
#!/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
30414093201713378043612608166064768844377641568960512000000000000
```

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:

```
#!/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 (caddr args))))
    (display (choose n m))
```

```
(newline)))
```

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
```

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@(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)`. See Section 6.23 [Other Languages], page 417, for more information on Guile's support for languages other than Scheme.

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 45, 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 iff value history is enabled.

`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 functionality. 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 supported 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 context 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* [REPL Command]
Run the optimizer on a piece of code and print the result.

`disassemble` *exp* [REPL Command]
Disassemble a compiled procedure.

`disassemble-file` *file* [REPL Command]
Disassemble a file.

4.4.4.5 Profile Commands

`time` *exp* [REPL Command]

Time execution.

`profile` *exp* [REPL Command]

Profile execution.

`trace` *exp* [REPL Command]

Trace execution.

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*] [REPL Command]

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*] [REPL Command]

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*] [REPL Command]

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*] [REPL Command]

Show a frame.

Show the selected frame. With an argument, select a frame by index, then show it.

`procedure` [REPL Command]

Print the procedure for the selected frame.

`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 10.3.3 [Stack Layout], page 740, 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 6.25.4.4 [Tracing Traps], page 444, 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 6.25.4 [Traps], page 440 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` [*key value*] [REPL Command]
List/show/set options.

`quit` [REPL Command]

Quit this session.

Current REPL options include:

`compile-options`
: The options used when compiling expressions entered at the REPL. See Section 6.17.5 [Compilation], page 350, 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.

`value-history`
: Whether value history is on or not. See Section 4.4.3 [Value History], page 46.

`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-set-default-option!` *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 45.

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 49, 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@(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@(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@(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@(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@(guile-user) [2]> ,q
scheme@(guile-user) [1]> ,q
scheme@(guile-user)>
```

Finally, as a word to the wise: hackers close their REPL prompts with `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 (`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 (`http://www.nongnu.org/geiser/`). Geiser complements Emacs' `scheme-mode` with tight integration to running Guile processes via a `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 `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 `guild` program.

```
$ guild compile -o foo.go foo.scm
wrote 'foo.go'
```

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 6.17.5 [Compilation], page 350, 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 6.17.7 [Load Paths], page 353, 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 6.22.1 [Build Config], page 412. If Guile 2.0 is installed on your system in `/usr/`, then `(%site-dir)` will be `/usr/share/guile/site/2.0`. 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 6.17.5 [Compilation], page 350, 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. Currently there is no procedure to get at this path, which is probably a bug. As in the previous example, if Guile 2.0 is installed on your system in `/usr/`, then the place to put compiled files for site packages will be `/usr/lib/guile/2.0/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. `Load Paths`, 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 6.22.1 [Build Config], page 412). Again, if Guile 2.0 is installed on your system in `/usr/`, then the extensions dir will be `/usr/lib/guile/2.0/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 *smobs*.

The Section 5.7 [Programming Overview], page 84 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 10.2 [Data Representation], page 729. 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 2.0.3, it will be compatible with some future version 2.0.7. We say in this case that 2.0 is the *effective version*, composed of the major and minor versions, in this case 2 and 0.

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 2.0, for example, you would invoke `pkg-config --cflags --libs guile-2.0` to get the compilation and linking flags necessary to link to version 2.0 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-2.0.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 54). Run `pkg-config guile-2.0 --variable=sitedir` to see its value. See Section 5.8.2 [Autoconf Macros], page 95, 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 6.20.4 [Modules and Extensions], page 386). Run `pkg-config guile-2.0 --variable=extensiondir` to see its value.

See the `pkg-config` man page, for more information, or its web site, `http://pkg-config.freedesktop.org/`. See Section 5.8 [Autoconf Support], page 95, 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 2.0:

```
pkg-config guile-2.0 --cflags
pkg-config guile-2.0 --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 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.

```
/* simple-guile.c --- how to start up the Guile
   interpreter from C code.  */

/* Get declarations for all the scm_ functions.  */
#include <libguile.h>

static void
inner_main (void *closure, int argc, char **argv)
{
  /* module initializations would go here */
  scm_shell (argc, argv);
}

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

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.

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

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

# Tell the C compiler where to find <libguile.h>
CFLAGS=`pkg-config --cflags guile-2.0`

# Tell the linker what libraries to use and where to find them.
LIBS=`pkg-config --libs guile-2.0`

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
```

If you are using the GNU Autoconf package to make your application more portable, Autoconf will settle many of the details in the Makefile above 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-2.0])

# 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 GUILE... 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`'.

```
#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-2.0` \
  -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 `LTDL_LIBRARY_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 58). 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 implementations 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.

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* and *flag*) that returns *a*+1 if *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 10.2 [Data Representation], page 729 to find out the details.

Some special SCM values are available to C code without needing to convert them from C values:

Scheme value	C representation
#f	SCM_BOOL_F
#t	SCM_BOOL_T
()	SCM_EOL

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 10.2 [Data Representation], page 729, 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*.

It is easy for Guile to remember all blocks of memory that it has allocated for use by Scheme values, but you need to help it with finding all Scheme values that are in use by C code.

You do this when writing a SMOB mark function, for example (see Section 5.5.4 [Garbage Collecting Smobs], page 76). By calling this function, the garbage collector learns about all references that your SMOB has to other `SCM` values.

Other references to `SCM` objects, such as global variables of type `SCM` or other random data structures in the heap that contain fields of type `SCM`, can be made visible to the garbage collector by calling the functions `scm_gc_protect` or `scm_permanent_object`. You normally use these functions for long lived objects such as a hash table that is stored in a global variable. For temporary references in local variables or function arguments, using these functions would be too expensive.

These references are handled differently: Local variables (and function arguments) of type `SCM` are automatically visible to the garbage collector. This works because the collector scans the stack for potential references to `SCM` objects and considers all referenced objects to be alive. The scanning considers each and every word of the stack, regardless of what it is actually used for, and then decides whether it could possibly be a reference to a `SCM` object. Thus, the scanning 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.

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 smob and work with that pointer directly. The reference to the `SCM` smob 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 smob 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 `scm_remember_upto_here_1` and its cousins. It will keep the compiler from overwriting the reference. For a typical example of its use, see Section 5.5.6 [Remembering During Operations], page 78.

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 `goto` statement, together with `if`. Loops done with `for`, `while` or `do` could in principle be rewritten with just `goto` and `if`. In Scheme, the primitive means for local control flow is the *function call* (together with `if`). 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.)

```
(define (foo-1 x)
  (bar-1 (1- x)))

(define (foo-2 x)
  (1- (bar-2 x)))
```

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 can not. 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.

```
(define (my-even? n)
  (cond ((zero? n) #t)
        (else (my-odd? (1- n)))))

(define (my-odd? n)
  (cond ((zero? n) #f)
        (else (my-even? (1- n)))))
```

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 *continuations* that have previously been captured using `call-with-current-continuation`. Guile also offers a slightly restricted set of functions, `catch` and `throw`, that can only be used for non-local exits. This restriction makes them more efficient. Error reporting (with the function `error`) is implemented by invoking `throw`, for example. The functions `catch` and `throw` belong to the topic of *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 `setjmp` and `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 `libguile`: it is best to assume that any `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 `with-output-to-port` needs to modify the global state so that `current-output-port` returns the port passed to `with-output-to-port`. The global output port needs to be reset to its previous value when `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 `dynamic-wind` function to arrange for the setting and resetting of the global state. C code can use the corresponding `scm_internal_dynamic_wind` function, or a `scm_dynwind_begin/scm_dynwind_end` pair together with suitable 'dynwind actions' (see Section 6.13.10 [Dynamic Wind], page 298).

Instead of coping with non-local control flow, you can also prevent it by erecting a *continuation barrier*, See Section 6.13.12 [Continuation Barriers], page 303. The function `scm_c_with_continuation_barrier`, 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 `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 *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 `scm_cons` can contain a safe point and when a signal handler is pending for your thread, calling `scm_cons` will run this handler and anything might happen, including a non-local exit although `scm_cons` 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 `scm_dynwind_block_asyncs`, for example. See See Section 6.21.2.1 [System asyncs], page 397.

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:

```
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 68.

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[1]; 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.

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:

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

  len = SCM_SIMPLE_VECTOR_LENGTH (vector);
  i = 0;
  while (i < len && scm_is_pair (list))
    {
      SCM_SIMPLE_VECTOR_SET (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

[1] 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 2.0.

list might get longer or shorter. For this reason, the loop must check both that it doesn't overrun the vector (SCM_SIMPLE_VECTOR_SET does no range-checking) and that it doesn't overrun the list (SCM_CAR and SCM_CDR likewise do no type checking).

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 simple 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_SIMPLE_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:

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

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

  return vector;
}
```

This version uses the type-checking and thread-robust functions scm_car and scm_cdr instead of the faster, but less robust macros 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-safeness 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 Types (Smobs)

Smobs are Guile's mechanism for adding new primitive types to the system. The term "smob" was coined by Aubrey Jaffer, who says it comes from "small object", referring to the fact that they are quite limited in size: they can hold just one pointer to a larger memory block plus 16 extra bits.

To define a new smob type, the programmer provides Guile with some essential information about the type — how to print it, how to garbage collect it, and so on — and Guile allocates a fresh type tag for it. The programmer can then use `scm_c_define_gsubr` to make a set of C functions visible to Scheme code that create and operate on these objects.

(You can find a complete version of the example code used in this section in the Guile distribution, in '`doc/example-smob`'. That directory includes a makefile and a suitable `main` function, so you can build a complete interactive Guile shell, extended with the datatypes described here.)

5.5.1 Describing a New Type

To define a new type, the programmer must write four functions to manage instances of the type:

`mark` Guile will apply this function to each instance of the new type it encounters during garbage collection. This function is responsible for telling the collector about any other `SCM` values that the object has stored. The default smob mark function does nothing. See Section 5.5.4 [Garbage Collecting Smobs], page 76, for more details.

`free` Guile will apply this function to each instance of the new type that is to be deallocated. The function should release all resources held by the object. This is analogous to the Java finalization method– it is invoked at an unspecified time (when garbage collection occurs) after the object is dead. The default free function frees the smob data (if the size of the struct passed to `scm_make_smob_type` is non-zero) using `scm_gc_free`. See Section 5.5.4 [Garbage Collecting Smobs], page 76, for more details.

This function operates while the heap is in an inconsistent state and must therefore be careful. See Section 6.8 [Smobs], page 233, for details about what this function is allowed to do.

`print` Guile will apply this function to each instance of the new type to print the value, as for `display` or `write`. The default print function prints `#<NAME ADDRESS>` where `NAME` is the first argument passed to `scm_make_smob_type`.

`equalp` If Scheme code asks the `equal?` function to compare two instances of the same smob type, Guile calls this function. It should return `SCM_BOOL_T` if *a* and *b* should be considered `equal?`, or `SCM_BOOL_F` otherwise. If `equalp` is `NULL`, `equal?` will assume that two instances of this type are never `equal?` unless they are `eq?`.

To actually register the new smob type, call `scm_make_smob_type`. It returns a value of type `scm_t_bits` which identifies the new smob type.

The four special functions described above are registered by calling one of `scm_set_smob_mark`, `scm_set_smob_free`, `scm_set_smob_print`, or `scm_set_smob_equalp`, as appropriate. Each function is intended to be used at most once per type, and the call should be placed immediately following the call to `scm_make_smob_type`.

There can only be at most 256 different smob types in the system. Instead of registering a huge number of smob types (for example, one for each relevant C struct in your application), it is sometimes better to register just one and implement a second layer of type dispatching on top of it. This second layer might use the 16 extra bits to extend its type, for example.

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

```
#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_t_bits image_tag;

void
init_image_type (void)
{
  image_tag = scm_make_smob_type ("image", sizeof (struct image));
  scm_set_smob_mark (image_tag, mark_image);
  scm_set_smob_free (image_tag, free_image);
  scm_set_smob_print (image_tag, print_image);
}
```

5.5.2 Creating Smob Instances

Normally, smobs can have one *immediate* word of data. This word stores either a pointer to an additional memory block that holds the real data, or it might hold the data itself when it fits. The word is large enough for a `SCM` value, a pointer to `void`, or an integer that fits into a `size_t` or `ssize_t`.

You can also create smobs that have two or three immediate words, and when these words suffice to store all data, it is more efficient to use these super-sized smobs instead of using a normal smob plus a memory block. See Section 5.5.7 [Double Smobs], page 79, for their discussion.

Guile provides functions for managing memory which are often helpful when implementing smobs. See Section 6.18.2 [Memory Blocks], page 359.

To retrieve the immediate word of a smob, you use the macro `SCM_SMOB_DATA`. It can be set with `SCM_SET_SMOB_DATA`. The 16 extra bits can be accessed with `SCM_SMOB_FLAGS` and `SCM_SET_SMOB_FLAGS`.

The two macros `SCM_SMOB_DATA` and `SCM_SET_SMOB_DATA` treat the immediate word as if it were of type `scm_t_bits`, which is an unsigned integer type large enough to hold a pointer to `void`. Thus you can use these macros to store arbitrary pointers in the smob word.

When you want to store a `SCM` value directly in the immediate word of a smob, you should use the macros `SCM_SMOB_OBJECT` and `SCM_SET_SMOB_OBJECT` to access it.

Creating a smob instance can be tricky when it consists of multiple steps that allocate resources and might fail. It is recommended that you go about creating a smob in the following way:

- Allocate the memory block for holding the data with `scm_gc_malloc`.
- Initialize it to a valid state without calling any functions that might cause a non-local exits. For example, initialize pointers to NULL. Also, do not store `SCM` values in it that must be protected. Initialize these fields with `SCM_BOOL_F`.

  A valid state is one that can be safely acted upon by the *mark* and *free* functions of your smob type.
- Create the smob using `SCM_NEWSMOB`, passing it the initialized memory block. (This step will always succeed.)
- Complete the initialization of the memory block by, for example, allocating additional resources and making it point to them.

This procedure ensures that the smob is in a valid state as soon as it exists, that all resources that are allocated for the smob are properly associated with it so that they can be properly freed, and that no `SCM` values that need to be protected are stored in it while the smob does not yet completely exist and thus can not protect them.

Continuing the example from above, if the global variable `image_tag` contains a tag returned by `scm_make_smob_type`, here is how we could construct a smob whose immediate word contains a pointer to a freshly allocated `struct image`:

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

  /* Step 1: Allocate the memory block.
   */
  image = (struct image *)
     scm_gc_malloc (sizeof (struct image), "image");

  /* Step 2: Initialize it with straight code.
   */
  image->width = width;
  image->height = height;
  image->pixels = NULL;
  image->name = SCM_BOOL_F;
  image->update_func = SCM_BOOL_F;

  /* Step 3: Create the smob.
   */
  SCM_NEWSMOB (smob, image_tag, image);

  /* Step 4: Finish the initialization.
   */
  image->name = name;
  image->pixels =
     scm_gc_malloc (width * height, "image pixels");

  return smob;
}
```

Let us look at what might happen when `make_image` is called.

The conversions of *s_width* and *s_height* to `int`s might fail and signal an error, thus causing a non-local exit. This is not a problem since no resources have been allocated yet that would have to be freed.

The allocation of *image* in step 1 might fail, but this is likewise no problem.

Step 2 can not exit non-locally. At the end of it, the *image* struct is in a valid state for the `mark_image` and `free_image` functions (see below).

Step 3 can not exit non-locally either. This is guaranteed by Guile. After it, *smob* contains a valid smob that is properly initialized and protected, and in turn can properly protect the Scheme values in its *image* struct.

But before the smob is completely created, `SCM_NEWSMOB` might cause the garbage collector to run. During this garbage collection, the `SCM` values in the *image* struct would be invisible to Guile. It only gets to know about them via the `mark_image` function, but that function can not yet do its job since the smob has not been created yet. Thus, it is important to not store `SCM` values in the *image* struct until after the smob has been created.

Step 4, finally, might fail and cause a non-local exit. In that case, the complete creation of the smob has not been successful, but it does nevertheless exist in a valid state. It will eventually be freed by the garbage collector, and all the resources that have been allocated for it will be correctly freed by `free_image`.

5.5.3 Type checking

Functions that operate on smobs should check that the passed `SCM` value indeed is a suitable smob before accessing its data. They can do this with `scm_assert_smob_type`.

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

```
SCM
clear_image (SCM image_smob)
{
  int area;
  struct image *image;

  scm_assert_smob_type (image_tag, image_smob);

  image = (struct image *) SCM_SMOB_DATA (image_smob);
  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);

  scm_remember_upto_here_1 (image_smob);

  return SCM_UNSPECIFIED;
}
```

See Section 5.5.6 [Remembering During Operations], page 78 for an explanation of the call to `scm_remember_upto_here_1`.

5.5.4 Garbage Collecting Smobs

Once a smob has been released to the tender mercies of the Scheme system, it must be prepared to survive garbage collection. Guile calls the *mark* and *free* functions of the smob to manage this.

As described in more detail elsewhere (see Section 10.2.4 [Conservative GC], page 733), every object in the Scheme system has a *mark bit*, which the garbage collector uses to tell live objects from dead ones. When collection starts, every object's mark bit is clear. The collector traces pointers through the heap, starting from objects known to be live, and sets the mark bit on each object it encounters. When it can find no more unmarked objects, the collector walks all objects, live and dead, frees those whose mark bits are still clear, and clears the mark bit on the others.

The two main portions of the collection are called the *mark phase*, during which the collector marks live objects, and the *sweep phase*, during which the collector frees all unmarked objects.

The mark bit of a smob lives in a special memory region. When the collector encounters a smob, it sets the smob's mark bit, and uses the smob's type tag to find the appropriate *mark* function for that smob. It then calls this *mark* function, passing it the smob as its only argument.

The *mark* function is responsible for marking any other Scheme objects the smob refers to. If it does not do so, the objects' mark bits will still be clear when the collector begins to sweep, and the collector will free them. If this occurs, it will probably break, or at least confuse, any code operating on the smob; the smob's `SCM` values will have become dangling references.

To mark an arbitrary Scheme object, the *mark* function calls `scm_gc_mark`.

Thus, here is how we might write `mark_image`:

```
SCM
mark_image (SCM image_smob)
{
  /* Mark the image's name and update function.  */
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_mark (image->name);
  scm_gc_mark (image->update_func);

  return SCM_BOOL_F;
}
```

Note that, even though the image's `update_func` could be an arbitrarily complex structure (representing a procedure and any values enclosed in its environment), `scm_gc_mark` will recurse as necessary to mark all its components. Because `scm_gc_mark` sets an object's mark bit before it recurses, it is not confused by circular structures.

As an optimization, the collector will mark whatever value is returned by the *mark* function; this helps limit depth of recursion during the mark phase. Thus, the code above should really be written as:

```
SCM
mark_image (SCM image_smob)
{
  /* Mark the image's name and update function.  */
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_mark (image->name);
  return image->update_func;
}
```

Finally, when the collector encounters an unmarked smob during the sweep phase, it uses the smob's tag to find the appropriate *free* function for the smob. It then calls that function, passing it the smob as its only argument.

The *free* function must release any resources used by the smob. However, it must not free objects managed by the collector; the collector will take care of them. For historical reasons, the return type of the *free* function should be `size_t`, an unsigned integral type; the *free* function should always return zero.

Here is how we might write the `free_image` function for the image smob type:

```
size_t
free_image (SCM image_smob)
{
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_free (image->pixels,
               image->width * image->height,
               "image pixels");
  scm_gc_free (image, sizeof (struct image), "image");

  return 0;
}
```

During the sweep phase, the garbage collector will clear the mark bits on all live objects. The code which implements a smob need not do this itself.

There is no way for smob code to be notified when collection is complete.

It is usually a good idea to minimize the amount of processing done during garbage collection; keep the *mark* and *free* functions very simple. Since collections occur at unpredictable times, it is easy for any unusual activity to interfere with normal code.

5.5.5 Garbage Collecting Simple Smobs

It is often useful to define very simple smob types — smobs which have no data to mark, other than the cell itself, or smobs whose immediate data word is simply an ordinary Scheme object, to be marked recursively. Guile provides some functions to handle these common cases; you can use this function as your smob type's *mark* function, if your smob's structure is simple enough.

If the smob refers to no other Scheme objects, then no action is necessary; the garbage collector has already marked the smob cell itself. In that case, you can use zero as your mark function.

If the smob refers to exactly one other Scheme object via its first immediate word, you can use `scm_markcdr` as its mark function. Its definition is simply:

```
SCM
scm_markcdr (SCM obj)
{
  return SCM_SMOB_OBJECT (obj);
}
```

5.5.6 Remembering During Operations

It's important that a smob is visible to the garbage collector whenever its contents are being accessed. Otherwise it could be freed while code is still using it.

For example, consider a procedure to convert image data to a list of pixel values.

```
SCM
image_to_list (SCM image_smob)
{
  struct image *image;
  SCM lst;
  int i;

  scm_assert_smob_type (image_tag, image_smob);

  image = (struct image *) SCM_SMOB_DATA (image_smob);
  lst = SCM_EOL;
  for (i = image->width * image->height - 1; i >= 0; i--)
    lst = scm_cons (scm_from_char (image->pixels[i]), lst);

  scm_remember_upto_here_1 (image_smob);
  return lst;
}
```

In the loop, only the `image` pointer is used and the C compiler has no reason to keep the `image_smob` value anywhere. If `scm_cons` results in a garbage collection, `image_smob` might not be on the stack or anywhere else and could be freed, leaving the loop accessing freed data. The use of `scm_remember_upto_here_1` prevents this, by creating a reference to `image_smob` after all data accesses.

There's no need to do the same for `lst`, since that's the return value and the compiler will certainly keep it in a register or somewhere throughout the routine.

The `clear_image` example previously shown (see Section 5.5.3 [Type checking], page 75) also used `scm_remember_upto_here_1` for this reason.

It's only in quite rare circumstances that a missing `scm_remember_upto_here_1` will bite, but when it happens the consequences are serious. Fortunately the rule is simple: whenever calling a Guile library function or doing something that might, ensure that the `SCM` of a smob is referenced past all accesses to its insides. Do this by adding an `scm_remember_upto_here_1` if there are no other references.

In a multi-threaded program, the rule is the same. As far as a given thread is concerned, a garbage collection still only occurs within a Guile library function, not at an arbitrary

time. (Guile waits for all threads to reach one of its library functions, and holds them there while the collector runs.)

5.5.7 Double Smobs

Smobs are called smob because they are small: they normally have only room for one `void*` or `SCM` value plus 16 bits. The reason for this is that smobs are directly implemented by using the low-level, two-word cells of Guile that are also used to implement pairs, for example. (see Section 10.2 [Data Representation], page 729 for the details.) One word of the two-word cells is used for `SCM_SMOB_DATA` (or `SCM_SMOB_OBJECT`), the other contains the 16-bit type tag and the 16 extra bits.

In addition to the fundamental two-word cells, Guile also has four-word cells, which are appropriately called *double cells*. You can use them for *double smobs* and get two more immediate words of type `scm_t_bits`.

A double smob is created with `SCM_NEWSMOB2` or `SCM_NEWSMOB3` instead of `SCM_NEWSMOB`. Its immediate words can be retrieved as `scm_t_bits` with `SCM_SMOB_DATA_2` and `SCM_SMOB_DATA_3` in addition to `SCM_SMOB_DATA`. Unsurprisingly, the words can be set to `scm_t_bits` values with `SCM_SET_SMOB_DATA_2` and `SCM_SET_SMOB_DATA_3`.

Of course there are also `SCM_SMOB_OBJECT_2`, `SCM_SMOB_OBJECT_3`, `SCM_SET_SMOB_OBJECT_2`, and `SCM_SET_SMOB_OBJECT_3`.

5.5.8 The Complete Example

Here is the complete text of the implementation of the image datatype, as presented in the sections above. We also provide a definition for the smob's *print* function, and make some objects and functions static, to clarify exactly what the surrounding code is using.

As mentioned above, you can find this code in the Guile distribution, in '`doc/example-smob`'. That directory includes a makefile and a suitable `main` function, so you can build a complete interactive Guile shell, extended with the datatypes described here.)

```
/* file "image-type.c" */

#include <stdlib.h>
#include <libguile.h>

static scm_t_bits image_tag;

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

  /* Step 1: Allocate the memory block.
   */
  image = (struct image *)
     scm_gc_malloc (sizeof (struct image), "image");

  /* Step 2: Initialize it with straight code.
   */
  image->width = width;
  image->height = height;
  image->pixels = NULL;
  image->name = SCM_BOOL_F;
  image->update_func = SCM_BOOL_F;

  /* Step 3: Create the smob.
   */
  SCM_NEWSMOB (smob, image_tag, image);

  /* Step 4: Finish the initialization.
   */
  image->name = name;
  image->pixels =
     scm_gc_malloc (width * height, "image pixels");

  return smob;
}

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

  scm_assert_smob_type (image_tag, image_smob);

  image = (struct image *) SCM_SMOB_DATA (image_smob);
  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);

  scm_remember_upto_here_1 (image_smob);

  return SCM_UNSPECIFIED;
}

static SCM
mark_image (SCM image_smob)
{
  /* Mark the image's name and update function.  */
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_mark (image->name);
  return image->update_func;
}

static size_t
free_image (SCM image_smob)
{
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_gc_free (image->pixels,
               image->width * image->height,
               "image pixels");
  scm_gc_free (image, sizeof (struct image), "image");

  return 0;
}

static int
print_image (SCM image_smob, SCM port, scm_print_state *pstate)
{
  struct image *image = (struct image *) SCM_SMOB_DATA (image_smob);

  scm_puts ("#<image ", port);
  scm_display (image->name, port);
  scm_puts (">", port);

  /* non-zero means success */
  return 1;
}
```

```
void
init_image_type (void)
{
  image_tag = scm_make_smob_type ("image", sizeof (struct image));
  scm_set_smob_mark (image_tag, mark_image);
  scm_set_smob_free (image_tag, free_image);
  scm_set_smob_print (image_tag, print_image);

  scm_c_define_gsubr ("clear-image", 1, 0, 0, clear_image);
  scm_c_define_gsubr ("make-image", 3, 0, 0, make_image);
}
```

Here is a sample build and interaction with the code from the ‘`example-smob`’ directory, on the author's machine:

```
zwingli:example-smob$ make CC=gcc
gcc ‘pkg-config --cflags guile-2.0‘ -c image-type.c -o image-type.o
gcc ‘pkg-config --cflags guile-2.0‘ -c myguile.c -o myguile.o
gcc image-type.o myguile.o ‘pkg-config --libs guile-2.0‘ -o myguile
zwingli:example-smob$ ./myguile
guile> make-image
#<primitive-procedure make-image>
guile> (define i (make-image "Whistler's Mother" 100 100))
guile> i
#<image Whistler's Mother>
guile> (clear-image i)
guile> (clear-image 4)
ERROR: In procedure clear-image in expression (clear-image 4):
ERROR: Wrong type (expecting image): 4
ABORT: (wrong-type-arg)

Type "(backtrace)" to get more information.
guile>
```

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 kind 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_smob),
            "Clear the image.")
{
  /* C code to clear the image in image_smob... */
}

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_smob`), 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:

```
(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 *SMOB* 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 *SMOBs* 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 the idea of SMOBs comes in, and with it 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.

```
(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 SMOB. The SMOB is what is passed into and out of Scheme code, and the Scheme-specific C structure inside the SMOB 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 SMOB. 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 SMOB 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 `struct dia_shape`):

- Define a new Scheme-specific structure that *points* to the underlying C structure:

```
struct dia_guile_shape
{
  struct dia_shape * c_shape;   /* NULL => deleted */
}
```

- Add a field to `struct dia_shape` that points to its `struct dia_guile_shape` if it has one —

```
struct dia_shape
{
  ...
  struct dia_guile_shape * guile_shape;
}
```

  — so that C code can set `guile_shape->c_shape` to NULL when the underlying shape is deleted.
- Wrap `struct dia_guile_shape` as a SMOB type.
- Whenever you need to represent a C shape onto the Scheme level, create a SMOB instance for it, and pass that.
- In primitive code that receives a shape SMOB instance, check the `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 SMOB 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 SMOB value, it calls the *SMOB free* function that was specified when defining the shape SMOB type. To maintain the correctness of the `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 `guile_shape` field to NULL.

For full documentation on defining and using SMOB types, see Section 5.5 [Defining New Types (Smobs)], page 71.

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 `SCM`, and whose body does whatever you want it to do. As an example, here is a possible implementation of the `square?` primitive:

```
static SCM square_p (SCM shape)
{
  struct dia_guile_shape * guile_shape;

  /* Check that arg is really a shape SMOB. */
  scm_assert_smob_type (shape_tag, shape);
```

```
    /* Access Scheme-specific shape structure. */
    guile_shape = SCM_SMOB_DATA (shape);

    /* 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 SMOB — to the Scheme-specific structure inside the SMOB, and thence to the underlying C structure for the shape.

In this code, `scm_assert_smob_type`, `SCM_SMOB_DATA`, and `scm_from_bool` are from the standard Guile API. We assume that `shape_tag` was given to us when we made the shape SMOB type, using `scm_make_smob_type`. The call to `scm_assert_smob_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 caught 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 237) 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 SMOB 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 SMOBs and have written primitive code as described in the preceding subsections, you also need to insert calls to declare your new SMOBs 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 SMOBs and primitives for the Scheme world.

For example, suppose that the original source code had a `dia_change_fill_pattern` function:

```
void dia_change_fill_pattern (struct dia_shape * shape,
                              struct dia_pattern * pattern)
{
  /* real pattern change work */
}
```

During initial Guile integration, you add a `change_fill_pattern` primitive for Scheme purposes, which accesses the underlying structures from its SMOB values and uses `dia_change_fill_pattern` to do the real work:

```
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;
}
```

At this point, it makes sense to keep `dia_change_fill_pattern` and `change_fill_pattern` separate, because `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 Gtk+ callback to be called.

But, if the code for creating buttons and registering their callbacks is moved into Scheme (using guile-gtk), it may become true that `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 `change_fill_pattern`, like this:

```
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;
}
```

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 SMOBs 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 SMOB 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 283).

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"[2] test plan for this application would aim to submit all pos-

[2] A *white box* test plan is one that incorporates knowledge of the internal design of the application under test.

sible 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.

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.[3] For example. . .

[3] 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.

- 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.

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`.

```
PKG_CHECK_MODULES([GUILE], [guile-2.0])
```

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_PROGS` [Autoconf Macro]

This macro looks for programs `guile`, `guile-config` and `guile-tools`, and sets variables *GUILE*, *GUILE_CONFIG* and *GUILE_TOOLS*, to their paths, respectively. If either of the first two is not found, signal error.

The variables are marked for substitution, as by `AC_SUBST`.

`GUILE_FLAGS` [Autoconf Macro]

This macro runs the `guile-config` script, installed with Guile, to find out where Guile's header files and libraries are installed. 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` for the Guile library itself, any libraries that Guile itself requires (like -lqthreads), and so on. It 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` [Autoconf Macro]

This looks for Guile's "site" directory, usually something like PREFIX/share/guile/site, and sets var *GUILE_SITE* to the path. Note that the var name is different from the macro name.

The variable is marked for substitution, as by `AC_SUBST`.

`GUILE_CHECK_RETVAL` *var check* [Autoconf Macro]

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` *var module featuretest description* [Autoconf Macro]

var is a shell variable name to be set to "yes" or "no". *module* is a list of symbols, like: (ice-9 common-list). *featuretest* is an expression acceptable to GUILE_CHECK, q.v. *description* is a present-tense verb phrase (passed to AC_MSG_CHECKING).

`GUILE_MODULE_AVAILABLE` *var module* [Autoconf Macro]

var is a shell variable name to be set to "yes" or "no". *module* is a list of symbols, like: (ice-9 common-list).

`GUILE_MODULE_REQUIRED` *symlist* [Autoconf Macro]

symlist is a list of symbols, WITHOUT surrounding parens, like: ice-9 common-list.

`GUILE_MODULE_EXPORTS` *var module modvar* [Autoconf Macro]

var is a shell variable to be set to "yes" or "no". *module* is a list of symbols, like: (ice-9 common-list). *modvar* is the Guile Scheme variable to check.

`GUILE_MODULE_REQUIRED_EXPORT` *module modvar* [Autoconf Macro]

module is a list of symbols, like: (ice-9 common-list). *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 '`configure.ac`', run `aclocal`, and finally, run `autoconf`. If your system doesn't have guile.m4 installed, place the desired macro definitions (`AC_DEFUN` forms) in '`acinclude.m4`', and `aclocal` will do the right thing.

Some of the macros can be used inside normal shell constructs: `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 `PKG_CHECK_MODULES` to set the vars `GUILE_CFLAGS` and `GUILE_LIBS`, which are automatically substituted in the Makefile.

```
In configure.ac:

  PKG_CHECK_MODULES([GUILE], [guile-2.0])

In Makefile.in:

  GUILE_CFLAGS  = @GUILE_CFLAGS@
  GUILE_LIBS = @GUILE_LIBS@

  myprog.o: myprog.c
          $(CC) -o $ $(GUILE_CFLAGS) $<
  myprog: myprog.o
          $(CC) -o $ $< $(GUILE_LIBS)
```

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 63.

`SCM` [C Type]

`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` [C Type]

`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` [C Type]

This is a signed integral type of the same size as `scm_t_bits`.

`scm_t_bits SCM_UNPACK` (*SCM x*) [C Macro]

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*) [C Macro]

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 `scm_with_guile` or `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 `scm_without_guile`, See Section 6.21.5 [Blocking], page 404.

Threads that are created by `call-with-new-thread` or `scm_spawn_thread` start out in guile mode so you don't need to initialize them.

`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 a 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.

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.

`void scm_boot_guile` (*int argc, char **argv, void (*main_func) (void *data, int argc, char **argv), void *data*) [C Function]

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 471).

`void scm_shell` (*int argc, char **argv*) [C Function]

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 82.

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*) [C Macro]

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*) [C Macro]

Normally, this macro expands into

```
static const char s_c_name[] = scheme_name;
SCM
c_name arglist
```

While snarfing, it causes

```
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*) [C Macro]
`SCM_GLOBAL_SYMBOL` (*c_name, scheme_name*) [C Macro]

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*) [C Macro]
`SCM_GLOBAL_KEYWORD` (*c_name, scheme_name*) [C Macro]

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*) [C Macro]
`SCM_GLOBAL_VARIABLE` (*c_name, scheme_name*) [C Macro]

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*) [C Macro]
`SCM_GLOBAL_VARIABLE_INIT` (*c_name, scheme_name, value*) [C Macro]

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 Simple Generic Data Types

This chapter describes those of Guile's simple data types which are primarily used for their role as items of generic data. By *simple* we mean data types that are not primarily used as containers to hold other data — i.e. pairs, lists, vectors and so on. For the documentation of such *compound* data types, see Section 6.7 [Compound Data Types], page 182.

6.6.1 Booleans

The two boolean values are `#t` for true and `#f` for false.

Boolean values are returned by predicate procedures, such as the general equality predicates `eq?`, `eqv?` and `equal?` (see Section 6.11.1 [Equality], page 265) and numerical and string comparison operators like `string=?` (see Section 6.6.5.7 [String Comparison], page 147) 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 281), 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`.

`not` *x* [Scheme Procedure]
`scm_not` (*x*) [C Function]
Return `#t` if *x* is `#f`, else return `#f`.

`boolean?` *obj* [Scheme Procedure]
`scm_boolean_p` (*obj*) [C Function]
Return `#t` if *obj* is either `#t` or `#f`, else return `#f`.

`SCM SCM_BOOL_T` [C Macro]
The `SCM` representation of the Scheme object `#t`.

`SCM SCM_BOOL_F` [C Macro]
The `SCM` representation of the Scheme object `#f`.

`int scm_is_true` (*SCM obj*) [C Function]
Return `0` if *obj* is `#f`, else return `1`.

`int scm_is_false` (*SCM obj*) [C Function]
Return `1` if *obj* is `#f`, else return `0`.

`int scm_is_bool` (*SCM obj*) [C Function]
Return `1` if *obj* is either `#t` or `#f`, else return `0`.

`SCM scm_from_bool` (*int val*) [C Function]
Return `#f` if *val* is `0`, else return `#t`.

`int scm_to_bool` (*SCM val*) [C Function]
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 p/q where p and q are integers; e.g. 9/16 works, but pi (an irrational number) doesn't. These include integers ($n/1$).

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 ($a + bi$, where a is the *real part*, b is the *imaginary part*, and i 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.

`number?` *obj* [Scheme Procedure]
`scm_number_p` (*obj*) [C Function]
: Return `#t` if *obj* is any kind of number, else `#f`.

For example:

```
(number? 3)
⇒ #t

(number? "hello there!")
⇒ #f

(define pi 3.141592654)
(number? pi)
⇒ #t
```

`int scm_is_number` (*SCM obj*) [C Function]
: 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.

```
(define (factorial n)
  (let loop ((n n) (product 1))
    (if (= n 0)
        product
        (loop (- n 1) (* product n)))))
```

```
(factorial 3)
⇒ 6

(factorial 20)
⇒ 2432902008176640000

(- (factorial 45))
⇒ -119622220865480194561963161495657715064383733760000000000
```

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 `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?` *x* [Scheme Procedure]
`scm_integer_p` (*x*) [C Function]

Return `#t` if *x* is an exact or inexact integer number, else `#f`.

```
(integer? 487)
⇒ #t

(integer? 3.0)
⇒ #t

(integer? -3.4)
⇒ #f

(integer? +inf.0)
```

⇒ `#t`

`int scm_is_integer` (*SCM x*) [C Function]

This is equivalent to `scm_is_true (scm_integer_p (x))`.

`scm_t_int8` [C Type]
`scm_t_uint8` [C Type]
`scm_t_int16` [C Type]
`scm_t_uint16` [C Type]
`scm_t_int32` [C Type]
`scm_t_uint32` [C Type]
`scm_t_int64` [C Type]
`scm_t_uint64` [C Type]
`scm_t_intmax` [C Type]
`scm_t_uintmax` [C Type]

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*) [C Function]
`int scm_is_unsigned_integer` (*SCM x, scm_t_uintmax min, scm_t_uintmax max*) [C Function]

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_ulong_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_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]

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_ulong_long`, `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*) [C Function]
`SCM scm_from_short` (*short x*) [C Function]
`SCM scm_from_ushort` (*unsigned short x*) [C Function]
`SCM scm_from_int` (*int x*) [C Function]
`SCM scm_from_uint` (*unsigned int x*) [C Function]
`SCM scm_from_long` (*long x*) [C Function]
`SCM scm_from_ulong` (*unsigned long x*) [C Function]
`SCM scm_from_long_long` (*long long x*) [C Function]
`SCM scm_from_ulong_long` (*unsigned long long x*) [C Function]
`SCM scm_from_size_t` (*size_t x*) [C Function]
`SCM scm_from_ssize_t` (*ssize_t x*) [C Function]
`SCM scm_from_int8` (*scm_t_int8 x*) [C Function]
`SCM scm_from_uint8` (*scm_t_uint8 x*) [C Function]
`SCM scm_from_int16` (*scm_t_int16 x*) [C Function]
`SCM scm_from_uint16` (*scm_t_uint16 x*) [C Function]
`SCM scm_from_int32` (*scm_t_int32 x*) [C Function]
`SCM scm_from_uint32` (*scm_t_uint32 x*) [C Function]
`SCM scm_from_int64` (*scm_t_int64 x*) [C Function]

SCM scm_from_uint64 (*scm_t_uint64 x*) [C Function]
SCM scm_from_intmax (*scm_t_intmax x*) [C Function]
SCM scm_from_uintmax (*scm_t_uintmax x*) [C Function]

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*) [C Function]

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*) [C Function]

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 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 π.

Guile can represent both exact and inexact rational numbers, but it cannot represent precise finite irrational numbers. Exact rationals are represented by storing the numerator 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:

```
1/2
-22/7
```

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:

```
0.34
-0.00000142857931198
-5648394822220000000000.0
4.0
```

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 100000000000000000. 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)` ⇒ `#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* [Scheme Procedure]
`scm_rational_p` (*x*) [C Function]

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* [Scheme Procedure]
`scm_rationalize` (*x*, *eps*) [C Function]

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* [Scheme Procedure]
`scm_inf_p` (*x*) [C Function]

Return `#t` if the real number *x* is '`+inf.0`' or '`-inf.0`'. Otherwise return `#f`.

`nan?` *x* [Scheme Procedure]
`scm_nan_p` (*x*) [C Function]

Return `#t` if the real number *x* is '`+nan.0`', or `#f` otherwise.

`finite?` *x* [Scheme Procedure]
`scm_finite_p` (*x*) [C Function]
Return `#t` if the real number *x* is neither infinite nor a NaN, `#f` otherwise.

`nan` [Scheme Procedure]
`scm_nan` () [C Function]
Return '`+nan.0`', a NaN value.

`inf` [Scheme Procedure]
`scm_inf` () [C Function]
Return '`+inf.0`', positive infinity.

`numerator` *x* [Scheme Procedure]
`scm_numerator` (*x*) [C Function]
Return the numerator of the rational number *x*.

`denominator` *x* [Scheme Procedure]
`scm_denominator` (*x*) [C Function]
Return the denominator of the rational number *x*.

`int scm_is_real` (*SCM val*) [C Function]
`int scm_is_rational` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_real_p (val))` and `scm_is_true (scm_rational_p (val))`, respectively.

`double scm_to_double` (*SCM val*) [C Function]
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*) [C Function]
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
⇒
3.0+4.0i

(* 3-8i 2.3+0.3i)
⇒
9.3-17.5i
```

Polar form can also be used, with an '`@`' between magnitude and angle,

```
1@3.141592 ⇒ -1.0      (approx)
-1@1.57079 ⇒ 0.0-1.0i  (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* [Scheme Procedure]
`scm_complex_p` (*z*) [C Function]

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*) [C Function]

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* [Scheme Procedure]
`scm_exact_p` (*z*) [C Function]

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*) [C Function]

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 approch to testing the exactness of a number is to use `scm_is_signed_integer` or `scm_is_unsigned_integer`.

`inexact?` *z* [Scheme Procedure]
`scm_inexact_p` (*z*) [C Function]

Return `#t` if the number *z* is inexact, `#f` else.

`int scm_is_inexact` (*SCM z*) [C Function]

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* [Scheme Procedure]
`scm_inexact_to_exact` (*z*) [C Function]

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* [Scheme Procedure]
`scm_exact_to_inexact` (*z*) [C Function]

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`
`#B` the integer is written in binary (base 2)

`#o`
`#O` the integer is written in octal (base 8)

`#d`
`#D` the integer is written in decimal (base 10)

`#x`
`#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`
`#E` the number is exact

`#i`
`#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+1i)
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* [Scheme Procedure]
`scm_odd_p` (*n*) [C Function]

Return `#t` if *n* is an odd number, `#f` otherwise.

`even?` *n* [Scheme Procedure]
`scm_even_p` (*n*) [C Function]

Return `#t` if *n* is an even number, `#f` otherwise.

`quotient` *n d* [Scheme Procedure]
`remainder` *n d* [Scheme Procedure]
`scm_quotient` (*n*, *d*) [C Function]
`scm_remainder` (*n*, *d*) [C Function]

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* [Scheme Procedure]
`scm_modulo` (*n*, *d*) [C Function]

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*... [Scheme Procedure]
`scm_gcd` (*x*, *y*) [C Function]

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*... [Scheme Procedure]
`scm_lcm` (*x*, *y*) [C Function]

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* [Scheme Procedure]
`scm_modulo_expt` (*n*, *k*, *m*) [C Function]

Return *n* raised to the integer exponent *k*, modulo *m*.

```
(modulo-expt 2 3 5)
   ⇒ 3
```

`exact-integer-sqrt` *k* [Scheme Procedure]
`void scm_exact_integer_sqrt` (*SCM k*, *SCM *s*, *SCM *r*) [C Function]

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.

`=` [Scheme Procedure]
`scm_num_eq_p` (*x*, *y*) [C Function]
Return `#t` if all parameters are numerically equal.

`<` [Scheme Procedure]
`scm_less_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically increasing.

`>` [Scheme Procedure]
`scm_gr_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically decreasing.

`<=` [Scheme Procedure]
`scm_leq_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically non-decreasing.

`>=` [Scheme Procedure]
`scm_geq_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically non-increasing.

`zero?` *z* [Scheme Procedure]
`scm_zero_p` (*z*) [C Function]
Return `#t` if *z* is an exact or inexact number equal to zero.

`positive?` *x* [Scheme Procedure]
`scm_positive_p` (*x*) [C Function]
Return `#t` if *x* is an exact or inexact number greater than zero.

`negative?` *x* [Scheme Procedure]
`scm_negative_p` (*x*) [C Function]
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 6.24.4 [Number Input and Output], page 423, for locale-dependent number parsing.

`number->string` *n* [*radix*] [Scheme Procedure]
`scm_number_to_string` (*n*, *radix*) [C Function]
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*] [Scheme Procedure]
`scm_string_to_number` (*string, radix*) [C Function]

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*) [C Function]

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* [Scheme Procedure]
`scm_make_rectangular` (*real_part, imaginary_part*) [C Function]

Return a complex number constructed of the given *real-part* and *imaginary-part* parts.

`make-polar` *mag ang* [Scheme Procedure]
`scm_make_polar` (*mag, ang*) [C Function]

Return the complex number *mag* * e^(i * *ang*).

`real-part` *z* [Scheme Procedure]
`scm_real_part` (*z*) [C Function]

Return the real part of the number *z*.

`imag-part` *z* [Scheme Procedure]
`scm_imag_part` (*z*) [C Function]

Return the imaginary part of the number *z*.

`magnitude` *z* [Scheme Procedure]
`scm_magnitude` (*z*) [C Function]

Return the magnitude of the number *z*. This is the same as `abs` for real arguments, but also allows complex numbers.

`angle` *z* [Scheme Procedure]
`scm_angle` (*z*) [C Function]

Return the angle of the complex number *z*.

`SCM scm_c_make_rectangular` (*double re, double im*) [C Function]
`SCM scm_c_make_polar` (*double x, double y*) [C Function]

Like `scm_make_rectangular` or `scm_make_polar`, respectively, but these functions take `double`s as their arguments.

`double scm_c_real_part` (*z*) [C Function]
`double scm_c_imag_part` (*z*) [C Function]

Returns the real or imaginary part of *z* as a `double`.

`double scm_c_magnitude` (*z*) [C Function]
`double scm_c_angle` (*z*) [C Function]

Returns the magnitude or angle of *z* as a `double`.

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* ... [Scheme Procedure]
`scm_sum` (*z1*, *z2*) [C Function]

Return the sum of all parameter values. Return 0 if called without any parameters.

`-` *z1 z2* ... [Scheme Procedure]
`scm_difference` (*z1*, *z2*) [C Function]

If called with one argument *z1*, -*z1* is returned. Otherwise the sum of all but the first argument are subtracted from the first argument.

`*` *z1* ... [Scheme Procedure]
`scm_product` (*z1*, *z2*) [C Function]

Return the product of all arguments. If called without arguments, 1 is returned.

`/` *z1 z2* ... [Scheme Procedure]
`scm_divide` (*z1*, *z2*) [C Function]

Divide the first argument by the product of the remaining arguments. If called with one argument *z1*, 1/*z1* is returned.

`1+` *z* [Scheme Procedure]
`scm_oneplus` (*z*) [C Function]

Return $z + 1$.

`1-` *z* [Scheme Procedure]
`scm_oneminus` (*z*) [C function]

Return $z - 1$.

`abs` *x* [Scheme Procedure]
`scm_abs` (*x*) [C Function]

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` *x1 x2* ... [Scheme Procedure]
`scm_max` (*x1*, *x2*) [C Function]

Return the maximum of all parameter values.

`min` *x1 x2* ... [Scheme Procedure]
`scm_min` (*x1*, *x2*) [C Function]

Return the minimum of all parameter values.

`truncate` *x* [Scheme Procedure]
`scm_truncate_number` (*x*) [C Function]

Round the inexact number *x* towards zero.

`round` *x* [Scheme Procedure]
`scm_round_number` (*x*) [C Function]

Round the inexact number *x* to the nearest integer. When exactly halfway between two integers, round to the even one.

`floor` *x* [Scheme Procedure]
`scm_floor` (*x*) [C Function]

Round the number *x* towards minus infinity.

`ceiling` *x* [Scheme Procedure]
`scm_ceiling` (*x*) [C Function]

Round the number *x* towards infinity.

`double scm_c_truncate` (*double x*) [C Function]
`double scm_c_round` (*double x*) [C Function]

Like `scm_truncate_number` or `scm_round_number`, respectively, but these functions take and return `double` values.

`euclidean/` *x y* [Scheme Procedure]
`euclidean-quotient` *x y* [Scheme Procedure]
`euclidean-remainder` *x y* [Scheme Procedure]
`void scm_euclidean_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_euclidean_quotient` (*SCM x, SCM y*) [C Function]
`SCM scm_euclidean_remainder` (*SCM x, SCM y*) [C Function]

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 $x = q * y + r$ and $0 <= r < |y|$. `euclidean/` returns both q and r, and is more efficient than computing each separately. Note that when $y > 0$, `euclidean-quotient` returns $floor(x/y)$, otherwise it returns $ceiling(x/y)$.

Note that these operators are equivalent to the R6RS operators `div`, `mod`, and `div-and-mod`.

```
(euclidean-quotient 123 10) ⇒ 12
(euclidean-remainder 123 10) ⇒ 3
(euclidean/ 123 10) ⇒ 12 and 3
(euclidean/ 123 -10) ⇒ -12 and 3
(euclidean/ -123 10) ⇒ -13 and 7
(euclidean/ -123 -10) ⇒ 13 and 7
(euclidean/ -123.2 -63.5) ⇒ 2.0 and 3.8
(euclidean/ 16/3 -10/7) ⇒ -3 and 22/21
```

`floor/` *x y* [Scheme Procedure]
`floor-quotient` *x y* [Scheme Procedure]
`floor-remainder` *x y* [Scheme Procedure]
`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]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `floor-quotient` returns the integer q and `floor-remainder` returns the real number r such that $q = floor(x/y)$ and $x = q * y + r$. `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, `floor-remainder` is equivalent to the R5RS integer-only operator `modulo`.

```
(floor-quotient 123 10) ⇒ 12
(floor-remainder 123 10) ⇒ 3
(floor/ 123 10) ⇒ 12 and 3
(floor/ 123 -10) ⇒ -13 and -7
(floor/ -123 10) ⇒ -13 and 7
(floor/ -123 -10) ⇒ 12 and -3
(floor/ -123.2 -63.5) ⇒ 1.0 and -59.7
(floor/ 16/3 -10/7) ⇒ -4 and -8/21
```

ceiling/ *x y* [Scheme Procedure]
ceiling-quotient *x y* [Scheme Procedure]
ceiling-remainder *x y* [Scheme Procedure]
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]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `ceiling-quotient` returns the integer q and `ceiling-remainder` returns the real number r such that $q = ceiling(x/y)$ and $x = q * y + r$. `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.

```
(ceiling-quotient 123 10) ⇒ 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 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 $-|y/2| <= r < |y/2|$.

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.

`sqrt` *z* [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,

```
(sqrt 9.0)       ⇒ 3.0
(sqrt -9.0)      ⇒ 0.0+3.0i
(sqrt 1.0+1.0i)  ⇒ 1.09868411346781+0.455089860562227i
(sqrt -1.0-1.0i) ⇒ 0.455089860562227-1.09868411346781i
```

`expt` *z1 z2* [Scheme Procedure]

Return *z1* raised to the power of *z2*.

`sin` *z* [Scheme Procedure]

Return the sine of *z*.

`cos` *z* [Scheme Procedure]

Return the cosine of *z*.

`tan` *z* [Scheme Procedure]
Return the tangent of *z*.

`asin` *z* [Scheme Procedure]
Return the arcsine of *z*.

`acos` *z* [Scheme Procedure]
Return the arccosine of *z*.

`atan` *z* [Scheme Procedure]
`atan` *y x* [Scheme Procedure]
Return the arctangent of *z*, or of y/x.

`exp` *z* [Scheme Procedure]
Return e to the power of *z*, where e is the base of natural logarithms (2.71828...).

`log` *z* [Scheme Procedure]
Return the natural logarithm of *z*.

`log10` *z* [Scheme Procedure]
Return the base 10 logarithm of *z*.

`sinh` *z* [Scheme Procedure]
Return the hyperbolic sine of *z*.

`cosh` *z* [Scheme Procedure]
Return the hyperbolic cosine of *z*.

`tanh` *z* [Scheme Procedure]
Return the hyperbolic tangent of *z*.

`asinh` *z* [Scheme Procedure]
Return the hyperbolic arcsine of *z*.

`acosh` *z* [Scheme Procedure]
Return the hyperbolic arccosine of *z*.

`atanh` *z* [Scheme Procedure]
Return the hyperbolic arctangent of *z*.

6.6.2.13 Bitwise Operations

For the following bitwise functions, negative numbers are treated as infinite precision twos-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` *n1 n2* ... [Scheme Procedure]
`scm_logand` (*n1*, *n2*) [C Function]
Return the bitwise AND of the integer arguments.

```
(logand) ⇒ -1
(logand 7) ⇒ 7
(logand #b111 #b011 #b001) ⇒ 1
```

`logior` *n1 n2 . . .* [Scheme Procedure]
`scm_logior` (*n1*, *n2*) [C Function]

Return the bitwise OR of the integer arguments.

```
(logior) ⇒ 0
(logior 7) ⇒ 7
(logior #b000 #b001 #b011) ⇒ 3
```

`logxor` *n1 n2 . . .* [Scheme Procedure]
`scm_loxor` (*n1*, *n2*) [C Function]

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.

```
(logxor) ⇒ 0
(logxor 7) ⇒ 7
(logxor #b000 #b001 #b011) ⇒ 2
(logxor #b000 #b001 #b011 #b011) ⇒ 1
```

`lognot` *n* [Scheme Procedure]
`scm_lognot` (*n*) [C Function]

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.

```
(number->string (lognot #b10000000) 2)
   ⇒ "-10000001"
(number->string (lognot #b0) 2)
   ⇒ "-1"
```

`logtest` *j k* [Scheme Procedure]
`scm_logtest` (*j*, *k*) [C Function]

Test whether *j* and *k* have any 1 bits in common. This is equivalent to `(not (zero? (logand j k)))`, but without actually calculating the `logand`, just testing for non-zero.

```
(logtest #b0100 #b1011) ⇒ #f
(logtest #b0100 #b0111) ⇒ #t
```

`logbit?` *index j* [Scheme Procedure]
`scm_logbit_p` (*index*, *j*) [C Function]

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 cnt* [Scheme Procedure]
`scm_ash` (*n*, *cnt*) [C Function]

Return *n* shifted left by *cnt* bits, or shifted right if *cnt* is negative. This is an "arithmetic" shift.

This is effectively a multiplication by 2^{cnt}, and when *cnt* is negative it's a division, rounded towards negative infinity. (Note that this is not the same rounding as `quotient` does.)

With *n* viewed as an infinite precision twos complement, `ash` means a left shift introducing zero bits, or a right shift dropping bits.

```
(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
```

`logcount` *n* [Scheme Procedure]
`scm_logcount` (*n*) [C Function]

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)
   ⇒ 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 twos complement form.

```
(integer-length #b10101010) ⇒ 8
(integer-length #b1111)     ⇒ 4
(integer-length 0)          ⇒ 0
(integer-length -1)         ⇒ 0
(integer-length -256)       ⇒ 8
(integer-length -257)       ⇒ 9
```

`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.

```
(integer-expt 2 5)   ⇒ 32
(integer-expt -3 3)  ⇒ -27
(integer-expt 5 -3)  ⇒ 1/125
(integer-expt 0 0)   ⇒ 1
```

`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.

```
(number->string (bit-extract #b1101101010 0 4) 2)
   ⇒ "1010"
(number->string (bit-extract #b1101101010 4 9) 2)
   ⇒ "10110"
```

6.6.2.14 Random Number Generation

Pseudo-random numbers are generated from a random state object, which can be created with `seed->random-state` or `datum->random-state`. An external representation (i.e. one which can written with `write` and read with `read`) of a random state object can be obtained via `random-state->datum`. The *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.

`random:exp` [*state*] [Scheme Procedure]
`scm_random_exp` (*state*) [C Function]

Return an inexact real in an exponential distribution with mean 1. For an exponential distribution with mean *u* use `(* u (random:exp))`.

`random:hollow-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_hollow_sphere_x` (*vect, state*) [C Function]

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* = `(vector-length vect)`, the coordinates are uniformly distributed over the surface of the unit n-sphere.

`random:normal` [*state*] [Scheme Procedure]
`scm_random_normal` (*state*) [C Function]

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* and standard deviation *d* use `(+ m (* d (random:normal)))`.

`random:normal-vector!` *vect* [*state*] [Scheme Procedure]
`scm_random_normal_vector_x` (*vect, state*) [C Function]

Fills *vect* with inexact real random numbers that are independent and standard normally distributed (i.e., with mean 0 and variance 1).

`random:solid-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_solid_sphere_x` (*vect*, *state*) [C Function]

Fills *vect* with inexact real random numbers the sum of whose squares is less than 1.0. Thinking of *vect* as coordinates in space of dimension n = `(vector-length vect)`, the coordinates are uniformly distributed within the unit n-sphere.

`random:uniform` [*state*] [Scheme Procedure]
`scm_random_uniform` (*state*) [C Function]

Return a uniformly distributed inexact real random number in [0,1).

`seed->random-state` *seed* [Scheme Procedure]
`scm_seed_to_random_state` (*seed*) [C Function]

Return a new random state using *seed*.

`datum->random-state` *datum* [Scheme Procedure]
`scm_datum_to_random_state` (*datum*) [C Function]

Return a new random state from *datum*, which should have been obtained by `random-state->datum`.

`random-state->datum` *state* [Scheme Procedure]
`scm_random_state_to_datum` (*state*) [C Function]

Return a datum representation of *state* that may be written out and read back with the Scheme reader.

`random-state-from-platform` [Scheme Procedure]
`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:

```
(map random (cdr (iota 19)))
⇒
(0 1 1 2 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 `#\name` where *name* is the name of the character that you want. Printable characters have their usual single character name; for example, `#\a` is a lower case `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 `#\` 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 `#\`.

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.

Character Name	Codepoint
`#\nul`	U+0000
`#\alarm`	u+0007
`#\backspace`	U+0008
`#\tab`	U+0009
`#\linefeed`	U+000A
`#\newline`	U+000A
`#\vtab`	U+000B
`#\page`	U+000C
`#\return`	U+000D
`#\esc`	U+001B
`#\space`	U+0020

`#\delete` U+007F

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.

0 = `#\nul`	1 = `#\soh`	2 = `#\stx`	3 = `#\etx`
4 = `#\eot`	5 = `#\enq`	6 = `#\ack`	7 = `#\bel`
8 = `#\bs`	9 = `#\ht`	10 = `#\lf`	11 = `#\vt`
12 = `#\ff`	13 = `#\cr`	14 = `#\so`	15 = `#\si`
16 = `#\dle`	17 = `#\dc1`	18 = `#\dc2`	19 = `#\dc3`
20 = `#\dc4`	21 = `#\nak`	22 = `#\syn`	23 = `#\etb`
24 = `#\can`	25 = `#\em`	26 = `#\sub`	27 = `#\esc`
28 = `#\fs`	29 = `#\gs`	30 = `#\rs`	31 = `#\us`
32 = `#\sp`			

The short name for the "delete" character (code point U+007F) is `#\del`.

There are also a few alternative names left over for compatibility with previous versions of Guile.

Alternate	Standard
`#\nl`	`#\newline`
`#\np`	`#\page`
`#\null`	`#\nul`

Characters may also be written using their code point values. They can be written with as 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.

`char?` *x* [Scheme Procedure]
`scm_char_p` (*x*) [C Function]

Return `#t` iff *x* is a character, else `#f`.

Fundamentally, the character comparison operations below are numeric comparisons of the character's code points.

`char=?` *x y* [Scheme Procedure]

Return `#t` iff code point of *x* is equal to the code point of *y*, else `#f`.

`char<?` *x y* [Scheme Procedure]

Return `#t` iff the code point of *x* is less than the code point of *y*, else `#f`.

`char<=?` *x y* [Scheme Procedure]

Return `#t` iff the code point of *x* is less than or equal to the code point of *y*, else `#f`.

`char>?` *x y* [Scheme Procedure]

Return `#t` iff the code point of *x* is greater than the code point of *y*, else `#f`.

`char>=?` *x y* [Scheme Procedure]

Return `#t` iff the code point of *x* is greater than or equal to the code point of *y*, else `#f`.

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.

`char-ci=?` *x y* [Scheme Procedure]

Return `#t` iff the case-folded code point of *x* is the same as the case-folded code point of *y*, else `#f`.

`char-ci<?` *x y* [Scheme Procedure]

Return `#t` iff the case-folded code point of *x* is less than the case-folded code point of *y*, else `#f`.

`char-ci<=?` *x y* [Scheme Procedure]

Return `#t` iff the case-folded code point of *x* is less than or equal to the case-folded code point of *y*, else `#f`.

`char-ci>?` *x y* [Scheme Procedure]

Return `#t` iff the case-folded code point of *x* is greater than the case-folded code point of *y*, else `#f`.

`char-ci>=?` *x y* [Scheme Procedure]

Return `#t` iff the case-folded code point of *x* is greater than or equal to the case-folded code point of *y*, else `#f`.

`char-alphabetic?` *chr* [Scheme Procedure]
`scm_char_alphabetic_p` (*chr*) [C Function]

Return `#t` iff *chr* is alphabetic, else `#f`.

`char-numeric?` *chr* [Scheme Procedure]
`scm_char_numeric_p` (*chr*) [C Function]

Return `#t` iff *chr* is numeric, else `#f`.

`char-whitespace?` *chr* [Scheme Procedure]
`scm_char_whitespace_p` (*chr*) [C Function]

Return `#t` iff *chr* is whitespace, else `#f`.

`char-upper-case?` *chr* [Scheme Procedure]
`scm_char_upper_case_p` (*chr*) [C Function]

Return `#t` iff *chr* is uppercase, else `#f`.

`char-lower-case?` *chr* [Scheme Procedure]
`scm_char_lower_case_p` (*chr*) [C Function]

Return `#t` iff *chr* is lowercase, else `#f`.

`char-is-both?` *chr* [Scheme Procedure]
`scm_char_is_both_p` (*chr*) [C Function]

Return `#t` iff *chr* is either uppercase or lowercase, else `#f`.

`char-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.

Lu	Uppercase letter	Pf	Final quote punctuation
Ll	Lowercase letter	Po	Other punctuation
Lt	Titlecase letter	Sm	Math symbol
Lm	Modifier letter	Sc	Currency symbol
Lo	Other letter	Sk	Modifier symbol
Mn	Non-spacing mark	So	Other symbol
Mc	Combining spacing mark	Zs	Space separator
Me	Enclosing mark	Zl	Line separator
Nd	Decimal digit number	Zp	Paragraph separator
Nl	Letter number	Cc	Control
No	Other number	Cf	Format
Pc	Connector punctuation	Cs	Surrogate
Pd	Dash punctuation	Co	Private use
Ps	Open punctuation	Cn	Unassigned
Pe	Close punctuation		
Pi	Initial quote punctuation		

`char->integer` *chr* [Scheme Procedure]
`scm_char_to_integer` (*chr*) [C Function]

Return the code point of *chr*.

`integer->char` *n* [Scheme Procedure]
`scm_integer_to_char` (*n*) [C Function]

Return the character that has code point *n*. The integer *n* must be a valid code point. Valid code points are in the ranges 0 to `#xD7FF` inclusive or `#xE000` to `#x10FFFF` inclusive.

`char-upcase` *chr* [Scheme Procedure]
`scm_char_upcase` (*chr*) [C Function]

Return the uppercase character version of *chr*.

`char-downcase` *chr* [Scheme Procedure]
`scm_char_downcase` (*chr*) [C Function]

Return the lowercase character version of *chr*.

`char-titlecase` *chr* [Scheme Procedure]
`scm_char_titlecase` (*chr*) [C Function]

Return the titlecase character version of *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).

`scm_t_wchar scm_c_upcase` (*scm_t_wchar c*) [C Function]
`scm_t_wchar scm_c_downcase` (*scm_t_wchar c*) [C Function]
`scm_t_wchar scm_c_titlecase` (*scm_t_wchar c*) [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 `scm_t_wchar` is a signed, 32-bit integer.

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 characters 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.

`char-set?` *obj* [Scheme Procedure]
`scm_char_set_p` (*obj*) [C Function]

Return `#t` if *obj* is a character set, `#f` otherwise.

`char-set=` *char_set* ... [Scheme Procedure]
`scm_char_set_eq` (*char_sets*) [C Function]

Return `#t` if all given character sets are equal.

`char-set<=` *char_set* ... [Scheme Procedure]
`scm_char_set_leq` (*char_sets*) [C Function]

Return `#t` if every character set *char_set*i is a subset of character set *char_set*i+1.

`char-set-hash` *cs* [*bound*] [Scheme Procedure]
`scm_char_set_hash` (*cs, bound*) [C Function]

Compute a hash value for the character set *cs*. If *bound* is given and non-zero, it restricts the returned value to the range 0 ... *bound* - 1.

6.6.4.2 Iterating Over Character Sets

Character set cursors are a means for iterating over the members of a character sets. 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.

`char-set-cursor` *cs* [Scheme Procedure]
`scm_char_set_cursor` (*cs*) [C Function]

Return a cursor into the character set *cs*.

`char-set-ref` *cs cursor* [Scheme Procedure]
`scm_char_set_ref` (*cs, cursor*) [C Function]

Return the character at the current cursor position *cursor* in the character set *cs*. It is an error to pass a cursor for which `end-of-char-set?` returns true.

`char-set-cursor-next` *cs cursor* [Scheme Procedure]
`scm_char_set_cursor_next` (*cs, cursor*) [C Function]

Advance the character set cursor *cursor* to the next character in the character set *cs*. It is an error if the cursor given satisfies `end-of-char-set?`.

`end-of-char-set?` *cursor* [Scheme Procedure]
`scm_end_of_char_set_p` (*cursor*) [C Function]

Return `#t` if *cursor* has reached the end of a character set, `#f` otherwise.

`char-set-fold` *kons knil cs* [Scheme Procedure]
`scm_char_set_fold` (*kons, knil, cs*) [C Function]

Fold the procedure *kons* over the character set *cs*, initializing it with *knil*.

`char-set-unfold` *p f g seed* [*base_cs*] [Scheme Procedure]
`scm_char_set_unfold` (*p, f, g, seed, base_cs*) [C Function]

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* [Scheme Procedure]
`scm_char_set_unfold_x` (*p, f, g, seed, base_cs*) [C Function]

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* [Scheme Procedure]
`scm_char_set_for_each` (*proc, cs*) [C Function]

Apply *proc* to every character in the character set *cs*. The return value is not specified.

`char-set-map` *proc cs* [Scheme Procedure]
`scm_char_set_map` (*proc, cs*) [C Function]

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* [Scheme Procedure]
`scm_char_set_copy` (*cs*) [C Function]
Return a newly allocated character set containing all characters in *cs*.

`char-set` *chr* ... [Scheme Procedure]
`scm_char_set` (*chrs*) [C Function]
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.

`->char-set` *x* [Scheme Procedure]
`scm_to_char_set` (*x*) [C Function]

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* [Scheme Procedure]

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* [Scheme Procedure]
`scm_char_set_size` (*cs*) [C Function]

Return the number of elements in character set *cs*.

`char-set-count` *pred cs* [Scheme Procedure]
`scm_char_set_count` (*pred, cs*) [C Function]

Return the number of the elements int the character set *cs* which satisfy the predicate *pred*.

`char-set->list` *cs* [Scheme Procedure]
`scm_char_set_to_list` (*cs*) [C Function]

Return a list containing the elements of the character set *cs*.

`char-set->string` *cs* [Scheme Procedure]
`scm_char_set_to_string` (*cs*) [C Function]

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* [Scheme Procedure]
`scm_char_set_contains_p` (*cs, ch*) [C Function]
Return `#t` iff the character *ch* is contained in the character set *cs*.

`char-set-every` *pred cs* [Scheme Procedure]
`scm_char_set_every` (*pred, cs*) [C Function]
Return a true value if every character in the character set *cs* satisfies the predicate *pred*.

`char-set-any` *pred cs* [Scheme Procedure]
`scm_char_set_any` (*pred, cs*) [C Function]
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* ... [Scheme Procedure]
`scm_char_set_adjoin` (*cs, chrs*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete` *cs chr* ... [Scheme Procedure]
`scm_char_set_delete` (*cs, chrs*) [C Function]
Delete all character arguments from the first argument, which must be a character set.

`char-set-adjoin!` *cs chr* ... [Scheme Procedure]
`scm_char_set_adjoin_x` (*cs, chrs*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete!` *cs chr* ... [Scheme Procedure]
`scm_char_set_delete_x` (*cs, chrs*) [C Function]
Delete all character arguments from the first argument, which must be a character set.

`char-set-complement` *cs* [Scheme Procedure]
`scm_char_set_complement` (*cs*) [C Function]
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* ... [Scheme Procedure]
`scm_char_set_union` (*char_sets*) [C Function]
Return the union of all argument character sets.

`char-set-intersection` *cs* ... [Scheme Procedure]
`scm_char_set_intersection` (*char_sets*) [C Function]
Return the intersection of all argument character sets.

`char-set-difference` *cs1 cs* ... [Scheme Procedure]
`scm_char_set_difference` (*cs1, char_sets*) [C Function]
Return the difference of all argument character sets.

`char-set-xor` *cs* ... [Scheme Procedure]
`scm_char_set_xor` (*char_sets*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection` *cs1 cs* ... [Scheme Procedure]
`scm_char_set_diff_plus_intersection` (*cs1, char_sets*) [C Function]
Return the difference and the intersection of all argument character sets.

`char-set-complement!` *cs* [Scheme Procedure]
`scm_char_set_complement_x` (*cs*) [C Function]
Return the complement of the character set *cs*.

`char-set-union!` *cs1 cs* ... [Scheme Procedure]
`scm_char_set_union_x` (*cs1, char_sets*) [C Function]
Return the union of all argument character sets.

`char-set-intersection!` *cs1 cs* ... [Scheme Procedure]
`scm_char_set_intersection_x` (*cs1, char_sets*) [C Function]
Return the intersection of all argument character sets.

`char-set-difference!` *cs1 cs* ... [Scheme Procedure]
`scm_char_set_difference_x` (*cs1, char_sets*) [C Function]
Return the difference of all argument character sets.

`char-set-xor!` *cs1 cs* ... [Scheme Procedure]
`scm_char_set_xor_x` (*cs1, char_sets*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection!` *cs1 cs2 cs* ... [Scheme Procedure]
`scm_char_set_diff_plus_intersection_x` (*cs1, cs2, char_sets*) [C Function]
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` [Scheme Variable]
`scm_char_set_lower_case` [C Variable]
All lower-case characters.

`char-set:upper-case` [Scheme Variable]
`scm_char_set_upper_case` [C Variable]
All upper-case characters.

`char-set:title-case` [Scheme Variable]
`scm_char_set_title_case` [C Variable]
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]
The union of `char-set:letter` and `char-set:digit`.

`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.

`char-set:empty` [Scheme Variable]
`scm_char_set_empty` [C Variable]
The empty character set.

`char-set:designated` [Scheme Variable]
`scm_char_set_designated` [C Variable]
This character set contains all designated code points. This includes all the code points to which Unicode has assigned a character or other meaning.

`char-set:full` [Scheme Variable]
`scm_char_set_full` [C Variable]
This character set contains all possible code points. This includes both designated and reserved code points.

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, 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).

`\a` Bell character (ASCII 7).

`\f` Formfeed character (ASCII 12).

`\n` Newline character (ASCII 10).

`\r` Carriage return character (ASCII 13).

`\t` Tab character (ASCII 9).

`\v` Vertical tab character (ASCII 11).

`\b` Backspace character (ASCII 8).

`\0` NUL character (ASCII 0).

`\` 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\
  bar"
⇒ "foo  bar"
(read-enable 'hungry-eol-escapes)
"foo\
  bar"
⇒ "foobar"
```

`\xHH` Character code given by two hexadecimal digits. For example `\x7f` for an ASCII DEL (127).

`\uHHHH` Character code given by four hexadecimal digits. For example `\u0100` for a capital A with macron (U+0100).

`\UHHHHHH` 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 6.17.2 [Scheme Read], page 346.

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.

```
(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 133) to check for any in that set, or a predicate procedure to call.

For a procedure, calls `(char_pred c)` 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 133) to check for every character being in that set, or a predicate procedure to call.

For a procedure, calls `(char_pred c)` 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 *end* − 1), if that point is reached, is a tail call and the return from that call is the return from `string-every`.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#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 144, for ways to create strings from existing strings.

`string` *char...* [Scheme Procedure]

Return a newly allocated string made from the given character arguments.

```
(string #\x #\y #\z) ⇒ "xyz"
(string)             ⇒ ""
```

`list->string` *lst* [Scheme Procedure]
`scm_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters.

```
(list->string '(#\a #\b #\c)) ⇒ "abc"
```

`reverse-list->string` *lst* [Scheme Procedure]
`scm_reverse_list_to_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters, in reverse order.

```
(reverse-list->string '(#\a #\B #\c)) ⇒ "cBa"
```

`make-string` *k* [*chr*] [Scheme Procedure]
`scm_make_string` (*k, chr*) [C Function]

Return a newly allocated string of length *k*. If *chr* is given, then all elements of the string are initialized to *chr*, otherwise the contents of the string are unspecified.

`SCM scm_c_make_string` (*size_t len, SCM chr*) [C Function]

Like `scm_make_string`, but expects the length as a `size_t`.

`string-tabulate` *proc len* [Scheme Procedure]
`scm_string_tabulate` (*proc, len*) [C Function]

proc is an integer->char procedure. Construct a string of size *len* by applying *proc* to each index to produce the corresponding string element. The order in which *proc* is applied to the indices is not specified.

`string-join` *ls* [*delimiter* [*grammar*]] [Scheme Procedure]
`scm_string_join` (*ls, delimiter, grammar*) [C Function]

Append the string in the string list *ls*, using the string *delimiter* as a delimiter between the elements of *ls*. *grammar* is a symbol which specifies how the delimiter is placed between the strings, and defaults to the symbol `infix`.

`infix` Insert the separator between list elements. An empty string will produce an empty list.

`string-infix`
Like `infix`, but will raise an error if given the empty list.

`suffix` Insert the separator after every list element.

`prefix` Insert the separator before each list element.

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.

`string->list` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_to_list` (*str, start, end*) [C Function]
`scm_string_to_list` (*str*) [C Function]
Convert the string *str* into a list of characters.

`string-split` *str chr* [Scheme Procedure]
`scm_string_split` (*str, chr*) [C Function]
Split the string *str* into a list of substrings delimited by appearances of the character *chr*. 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.

`string-length` *string* [Scheme Procedure]
`scm_string_length` (*string*) [C Function]
Return the number of characters in *string*.

`size_t scm_c_string_length` (*SCM str*) [C Function]
Return the number of characters in *str* as a `size_t`.

`string-ref` *str k* [Scheme Procedure]
`scm_string_ref` (*str, k*) [C Function]
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` *s n* [Scheme Procedure]
`scm_string_drop_right` (*s, n*) [C Function]

Return all but the last *n* characters of *s*.

`string-pad` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`string-pad-right` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`scm_string_pad` (*s, len, chr, start, end*) [C Function]
`scm_string_pad_right` (*s, len, chr, start, end*) [C Function]

Take characters *start* to *end* from the string *s* and either pad with *chr* or truncate them to give *len* characters.

`string-pad` pads or truncates on the left, so for example

```
(string-pad "x" 3)     ⇒ "  x"
(string-pad "abcde" 3) ⇒ "cde"
```

`string-pad-right` pads or truncates on the right, so for example

```
(string-pad-right "x" 3)     ⇒ "x  "
(string-pad-right "abcde" 3) ⇒ "abc"
```

`string-trim` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-right` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-both` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`scm_string_trim` (*s, char_pred, start, end*) [C Function]
`scm_string_trim_right` (*s, char_pred, start, end*) [C Function]
`scm_string_trim_both` (*s, char_pred, start, end*) [C Function]

Trim occurrences of *char_pred* from the ends of *s*.

`string-trim` trims *char_pred* characters from the left (start) of the string, `string-trim-right` trims them from the right (end) of the string, `string-trim-both` trims from both ends.

char_pred can be a character, a character set, or a predicate procedure to call on each character. If *char_pred* is not given the default is whitespace as per `char-set:whitespace` (see Section 6.6.4.6 [Standard Character Sets], page 138).

```
(string-trim " x ")              ⇒ "x "
(string-trim-right "banana" #\a) ⇒ "banan"
(string-trim-both ".,xy:;" char-set:punctuation)
                   ⇒ "xy"
(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.

`string-set!` *str k chr* [Scheme Procedure]
`scm_string_set_x` (*str, k, chr*) [C Function]

Store *chr* in element *k* of *str* and return an unspecified value. *k* must be a valid index of *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 6.24.2 [Text Collation], page 422, 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 . . .* [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 or equal to *str_i+1*.

`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 greater than *str_i+1*.

`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 greater than or equal to *str_i+1*.

`string-ci=?` *s1 s2 s3 . . .* [Scheme Procedure]

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 . . .* [Scheme Procedure]

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 . . .* [Scheme Procedure]

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 . . .* [Scheme Procedure]

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 . . .* [Scheme Procedure]

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 <= *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*]]]] [Scheme Procedure]
`scm_string_compare_ci` (*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 <= *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*]]]] [Scheme Procedure]
`scm_string_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are not equal, a true value otherwise.

`string<>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are equal, a true value otherwise.

`string<` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise.

`string>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise.

`string<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_le` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater to *s2*, a true value otherwise.

`string>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less to *s2*, a true value otherwise.

`string-ci=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

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*]]]] [Scheme Procedure]
`scm_string_ci_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

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*]]]] [Scheme Procedure]
`scm_string_ci_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

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*]]]] [Scheme Procedure]
`scm_string_ci_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_le` (*s1, s2, start1, end1, start2, end2*) [C Function]

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*]]]] [Scheme Procedure]
`scm_string_ci_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]

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*]]] [Scheme Procedure]
`scm_substring_hash` (*s, bound, start, end*) [C Function]

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*]]] [Scheme Procedure]
`scm_substring_hash_ci` (*s, bound, start, end*) [C Function]

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.

`string-normalize-nfd` *s* [Scheme Procedure]
`scm_string_normalize_nfd` (*s*) [C Function]
Return the `NFD` normalized form of *s*.

`string-normalize-nfkd` *s* [Scheme Procedure]
`scm_string_normalize_nfkd` (*s*) [C Function]
Return the `NFKD` normalized form of *s*.

`string-normalize-nfc` *s* [Scheme Procedure]
`scm_string_normalize_nfc` (*s*) [C Function]
Return the `NFC` normalized form of *s*.

`string-normalize-nfkc` *s* [Scheme Procedure]
`scm_string_normalize_nfkc` (*s*) [C Function]
Return the `NFKC` normalized form of *s*.

6.6.5.8 String Searching

`string-index` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from left to right, returning the index of the first 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 *char_pred*, if it is a character set.

Return `#f` if no match is found.

`string-rindex` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_rindex` (*s, char_pred, start, end*) [C Function]
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.

Return `#f` if no match is found.

`string-prefix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common prefix of the two strings.

`string-prefix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common prefix of the two strings, ignoring character case.

`string-suffix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common suffix of the two strings.

`string-suffix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the length of the longest common suffix of the two strings, ignoring character case.

`string-prefix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a prefix of *s2*?

`string-prefix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a prefix of *s2*, ignoring character case?

`string-suffix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a suffix of *s2*?

`string-suffix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]

Is *s1* a suffix of *s2*, ignoring character case?

`string-index-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index_right` (*s, char_pred, start, end*) [C Function]

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.

Return `#f` if no match is found.

`string-skip` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip` (*s, char_pred, start, end*) [C Function]

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*]] [Scheme Procedure]
`scm_string_skip_right` (*s, char_pred, start, end*) [C Function]

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]
`scm_string_count` (*s, char_pred, start, end*) [C Function]

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]
`scm_string_contains` (*s1, s2, start1, end1, start2, end2*) [C Function]

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]
`scm_string_contains_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]

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 6.24.3 [Character Case Mapping], page 423, for locale-dependent case conversions.

`string-upcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase` (*str, start, end*) [C Function]
`scm_string_upcase` (*str*) [C Function]

Upcase every character in `str`.

`string-upcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase_x` (*str, start, end*) [C Function]
`scm_string_upcase_x` (*str*) [C Function]

Destructively upcase every character in `str`.

```
(string-upcase! y)
⇒ "ARRDEFG"
y
⇒ "ARRDEFG"
```

`string-downcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase` (*str, start, end*) [C Function]
`scm_string_downcase` (*str*) [C Function]

Downcase every character in *str*.

`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,

```
(define str (string-copy "studly"))
(string-for-each-index
    (lambda (i)
      (string-set! str i
        ((if (even? i) char-upcase char-downcase)
         (string-ref str i))))
    str)
str ⇒ "StUdLy"
```

`string-fold` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold` (*kons, knil, s, start, end*) [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*]] [Scheme Procedure]
`scm_string_fold_right` (*kons, knil, s, start, end*) [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*]] [Scheme Procedure]
`scm_string_unfold` (*p, f, g, seed, base, make_final*) [C Function]

- *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 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*]] [Scheme Procedure]
`scm_string_unfold_right` (*p, f, g, seed, base, make_final*) [C Function]

- *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 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.

- *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]
`scm_xsubstring` (*s, from, to, start, end*) [C Function]

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 sfrom* [*sto* [*start* [*end*]]] [Scheme Procedure]
`scm_string_xcopy_x` (*target, tstart, s, sfrom, sto, start, end*) [C Function]

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]
`scm_string_replace` (*s1, s2, start1, end1, start2, end2*) [C Function]

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]
`scm_string_tokenize` (*s, token_set, start, end*) [C Function]

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]
`scm_string_filter` (*char_pred, s, start, end*) [C Function]

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]
`scm_string_delete` (*char_pred, s, start, end*) [C Function]

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 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 never get to see the bytes, only the characters.

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 298.

`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 source code. To convert C string constants, use `scm_from_latin1_string`, `scm_from_utf8_string` or `scm_from_utf32_string`.

`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 298.

For `scm_to_locale_string`, the returned string is null-terminated and an error is signalled when *str* contains `#\nul` characters.

For `scm_to_locale_stringn` and *lenp* not `NULL`, *str* might contain `#\nul` 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 304, 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 `SCM_FAILED_CONVERSION_ERROR` and `SCM_FAILED_CONVERSION_QUESTION_MARK` can be used when converting C strings to Scheme.

`char *scm_to_stringn` (*SCM str, size_t *lenp, const char *encoding, scm_t_string_failed_conversion_handler handler*) [C Function]

This function returns a newly allocated C string from the Guile string *str*. The length of the returned string in bytes will be returned in *lenp*. The character encoding of the C string is passed as the ASCII, null-terminated C string *encoding*. The *handler* parameter gives a strategy for dealing with characters that cannot be converted into *encoding*.

If *lenp* is `NULL`, this function will return a null-terminated C string. It will throw an error if the string contains a null character.

`SCM scm_from_stringn` (*const char *str, size_t len, const char *encoding, scm_t_string_failed_conversion_handler handler*) [C Function]

This function returns a scheme string from the C string *str*. The length in bytes of the C string is input as *len*. The encoding of the C string is passed as the ASCII, null-terminated C string `encoding`. The *handler* parameters suggests a strategy for dealing with unconvertable characters.

The following conversion functions are provided as a convenience for the most commonly used encodings.

`SCM scm_from_latin1_string` (*const char *str*) [C Function]
`SCM scm_from_utf8_string` (*const char *str*) [C Function]
`SCM scm_from_utf32_string` (*const scm_t_wchar *str*) [C Function]

Return a scheme string from the null-terminated C string *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.

`SCM scm_from_latin1_stringn` (*const char *str, size_t len*) [C Function]
`SCM scm_from_utf8_stringn` (*const char *str, size_t len*) [C Function]
`SCM scm_from_utf32_stringn` (*const scm_t_wchar *str, size_t len*) [C Function]

Return a scheme string from C string *str*, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded, of length *len*. *len* is the number of bytes pointed to by *str* for `scm_from_latin1_stringn` and `scm_from_utf8_stringn`; it is the number of elements (code points) in *str* in the case of `scm_from_utf32_stringn`.

`char *scm_to_latin1_stringn` (*SCM str, size_t *lenp*) [C function]
`char *scm_to_utf8_stringn` (*SCM str, size_t *lenp*) [C function]
`scm_t_wchar *scm_to_utf32_stringn` (*SCM str, size_t *lenp*) [C function]

Return a newly allocated, ISO-8859-1-, UTF-8-, or UTF-32-encoded C string from Scheme string *str*. An error is thrown when *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`.

6.6.5.14 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.

`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.

`%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 Bytevectors

A *bytevector* 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/`). 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 of the R6RS (see Section 6.14.10 [R6RS I/O Ports], page 319).

6.6.6.1 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[1].

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.

`native-endianness` [Scheme Procedure]
`scm_native_endianness ()` [C Function]

Return a value denoting the native endianness of the host machine.

`endianness` *symbol* [Scheme Macro]

Return an object denoting the endianness specified by *symbol*. If *symbol* is neither `big` nor `little` then an error is raised at expand-time.

`scm_endianness_big` [C Variable]
`scm_endianness_little` [C Variable]

The objects denoting big- and little-endianness, respectively.

6.6.6.2 Manipulating Bytevectors

Bytevectors can be created, copied, and analyzed with the following procedures and C functions.

[1] Big-endian and little-endian are the most common "endiannesses", 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*).

`make-bytevector` *len* [*fill*] [Scheme Procedure]
`scm_make_bytevector` (*len*, *fill*) [C Function]
`scm_c_make_bytevector` (*size_t len*) [C Function]

Return a new bytevector of *len* bytes. Optionally, if *fill* is given, fill it with *fill*; *fill* must be in the range [-128,255].

`bytevector?` *obj* [Scheme Procedure]
`scm_bytevector_p` (*obj*) [C Function]

Return true if *obj* is a bytevector.

`int scm_is_bytevector` (*SCM obj*) [C Function]

Equivalent to `scm_is_true (scm_bytevector_p (obj))`.

`bytevector-length` *bv* [Scheme Procedure]
`scm_bytevector_length` (*bv*) [C Function]

Return the length in bytes of bytevector *bv*.

`size_t scm_c_bytevector_length` (*SCM bv*) [C Function]

Likewise, return the length in bytes of bytevector *bv*.

`bytevector=?` *bv1 bv2* [Scheme Procedure]
`scm_bytevector_eq_p` (*bv1*, *bv2*) [C Function]

Return is *bv1* equals to *bv2*—i.e., if they have the same length and contents.

`bytevector-fill!` *bv fill* [Scheme Procedure]
`scm_bytevector_fill_x` (*bv*, *fill*) [C Function]

Fill bytevector *bv* with *fill*, a byte.

`bytevector-copy!` *source source-start target target-start len* [Scheme Procedure]
`scm_bytevector_copy_x` (*source*, *source_start*, *target*, *target_start*, *len*) [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` *bv* [Scheme Procedure]
`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*.

6.6.6.3 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.

```
(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* [Scheme Procedure]
`scm_bytevector_uint_ref` (*bv, index, endianness, size*) [C Function]

Return the *size*-byte long unsigned integer at index *index* in *bv*, decoded according to *endianness*.

`bytevector-sint-ref` *bv index endianness size* [Scheme Procedure]
`scm_bytevector_sint_ref` (*bv, index, endianness, size*) [C Function]

Return the *size*-byte long signed integer at index *index* in *bv*, decoded according to *endianness*.

`bytevector-uint-set!` *bv index value endianness size* [Scheme Procedure]
`scm_bytevector_uint_set_x` (*bv, index, value, endianness, size*) [C Function]

Set the *size*-byte long unsigned integer at *index* to *value*, encoded according to *endianness*.

`bytevector-sint-set!` *bv index value endianness size* [Scheme Procedure]
`scm_bytevector_sint_set_x` (*bv, index, value, endianness, size*) [C Function]

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* [Scheme Procedure]
`bytevector-s8-ref` *bv index* [Scheme Procedure]
`bytevector-u16-ref` *bv index endianness* [Scheme Procedure]

`bytevector-s16-ref` *bv index endianness* [Scheme Procedure]
`bytevector-u32-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s32-ref` *bv index endianness* [Scheme Procedure]
`bytevector-u64-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s64-ref` *bv index endianness* [Scheme Procedure]
`scm_bytevector_u8_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s8_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u16_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s16_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_u32_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s32_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_u64_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s64_ref` (*bv*, *index*, *endianness*) [C Function]

Return the unsigned *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) from *bv* at *index*, decoded according to *endianness*.

`bytevector-u8-set!` *bv index value* [Scheme Procedure]
`bytevector-s8-set!` *bv index value* [Scheme Procedure]
`bytevector-u16-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s16-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-u32-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s32-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-u64-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s64-set!` *bv index value endianness* [Scheme Procedure]
`scm_bytevector_u8_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s8_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u16_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s16_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_u32_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s32_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_u64_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s64_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]

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` accessors, for obvious reasons):

`bytevector-u16-native-ref` *bv index* [Scheme Procedure]
`bytevector-s16-native-ref` *bv index* [Scheme Procedure]
`bytevector-u32-native-ref` *bv index* [Scheme Procedure]
`bytevector-s32-native-ref` *bv index* [Scheme Procedure]
`bytevector-u64-native-ref` *bv index* [Scheme Procedure]
`bytevector-s64-native-ref` *bv index* [Scheme Procedure]
`scm_bytevector_u16_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s16_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u32_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s32_native_ref` (*bv*, *index*) [C Function]

`scm_bytevector_u64_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s64_native_ref` (*bv*, *index*) [C Function]

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* [Scheme Procedure]
`bytevector-s16-native-set!` *bv index value* [Scheme Procedure]
`bytevector-u32-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s32-native-set!` *bv index value* [Scheme Procedure]
`bytevector-u64-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s64-native-set!` *bv index value* [Scheme Procedure]
`scm_bytevector_u16_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s16_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u32_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s32_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u64_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s64_native_set_x` (*bv*, *index*, *value*) [C Function]

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.

6.6.6.4 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*.

6.6.6.5 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*) [C Function]

Store real number *value* in *bv* at *index* according to *endianness*.

Specialized procedures are also available:

`bytevector-ieee-single-native-ref` *bv index* [Scheme Procedure]
`bytevector-ieee-double-native-ref` *bv index* [Scheme Procedure]
`scm_bytevector_ieee_single_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_ieee_double_native_ref` (*bv*, *index*) [C Function]

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* [Scheme Procedure]
`bytevector-ieee-double-native-set!` *bv index value* [Scheme Procedure]
`scm_bytevector_ieee_single_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_ieee_double_native_set_x` (*bv*, *index*, *value*) [C Function]

Store real number *value* in *bv* at *index* according to the host's native endianness.

6.6.6.6 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.

```
(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` *str* [Scheme Procedure]
`string->utf16` *str* [*endianness*] [Scheme Procedure]

`string->utf32` *str* [*endianness*] [Scheme Procedure]
`scm_string_to_utf8` (*str*) [C Function]
`scm_string_to_utf16` (*str, endianness*) [C Function]
`scm_string_to_utf32` (*str, endianness*) [C Function]

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* [Scheme Procedure]
`utf16->string` *utf* [*endianness*] [Scheme Procedure]
`utf32->string` *utf* [*endianness*] [Scheme Procedure]
`scm_utf8_to_string` (*utf*) [C Function]
`scm_utf16_to_string` (*utf, endianness*) [C Function]
`scm_utf32_to_string` (*utf, endianness*) [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.

6.6.6.7 Accessing Bytevectors with the Generalized Vector API

As an extension to the R6RS, Guile allows bytevectors to be manipulated with the *generalized vector* procedures (see Section 6.7.5 [Generalized Vectors], page 197). This also allows bytevectors to be accessed using the generic *array* procedures (see Section 6.7.6.2 [Array Procedures], page 199). When using these APIs, bytes are accessed one at a time as 8-bit unsigned integers:

```
(define bv #vu8(0 1 2 3))

(generalized-vector? bv)
⇒ #t

(generalized-vector-ref bv 2)
⇒ 2

(generalized-vector-set! bv 2 77)
(array-ref bv 2)
⇒ 77

(array-type bv)
⇒ vu8
```

6.6.6.8 Accessing Bytevectors with the SRFI-4 API

Bytevectors may also be accessed with the SRFI-4 API. See Section 7.5.5.4 [SRFI-4 and Bytevectors], page 557, for more information.

6.6.7 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 6.6.7.3 [Symbol Variables], page 171).

6.6.7.1 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 car) 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 car) 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 car) '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 car1)
⇒
(red manual unleaded power-steering)

(if (memq 'power-steering (properties-of car1))
    (display "Unfit people can drive this car.\n")
    (display "You'll need strong arms to drive this car!\n"))
⊣
Unfit people can drive this car.
```

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.

6.6.7.2 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:

```
(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:

```
(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 6.7.11 [Association Lists], page 220) to extract or change individual pieces of information:

```
(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:

`symbol-hash` *symbol* [Scheme Procedure]
`scm_symbol_hash` (*symbol*) [C Function]

Return a hash value for *symbol*.

See Section 6.7.13 [Hash Tables], page 228 for information about hash tables in general, and for why you might choose to use a hash table rather than an association list.

6.6.7.3 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 275, for how associations between symbols and variables are created, and Section 6.19 [Modules], page 364, for how those associations are affected by Guile's module system.

6.6.7.4 Operations Related to Symbols

Given any Scheme value, you can determine whether it is a symbol using the `symbol?` primitive:

`symbol?` *obj* [Scheme Procedure]
`scm_symbol_p` (*obj*) [C Function]
Return `#t` if *obj* is a symbol, otherwise return `#f`.

`int scm_is_symbol` (*SCM val*) [C Function]
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:

`symbol->string` *s* [Scheme Procedure]
`scm_symbol_to_string` (*s*) [C Function]
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.

```
(symbol #\x #\y #\z) ⇒ xyz
```

`list->symbol` *lst* [Scheme Procedure]
Return a newly allocated symbol made from a list of characters.

```
(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* ....

```
(let ((h 'hello))
  (symbol-append h 'world))
⇒ helloworld
```

`string->symbol` *string* [Scheme Procedure]
`scm_string_to_symbol` (*string*) [C Function]

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]
`scm_string_ci_to_symbol` (*str*) [C Function]

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:

```
(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_from_latin1_symbol (const` *char *name*`)` [C Function]
`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_from_locale_symbol (const` *char *name*`)` [C Function]
`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 source code. In such cases, use `scm_from_latin1_symbol` or `scm_from_utf8_symbol`.

`SCM scm_take_locale_symbol (`*char *str*`)` [C Function]
`SCM scm_take_locale_symboln (`*char *str, size_t len*`)` [C Function]

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*`)` [C Function]

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*`]` [Scheme Procedure]
`scm_gensym (`*prefix*`)` [C Function]

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 6.6.7.7 [Symbol Uninterned], page 176), though they can't be usefully written out and read back in.

6.6.7.5 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* [Scheme Procedure]
`scm_symbol_fref` (*symbol*) [C Function]
Return the contents of *symbol*'s *function slot*.

`symbol-fset!` *symbol value* [Scheme Procedure]
`scm_symbol_fset_x` (*symbol, value*) [C Function]
Set the contents of *symbol*'s function slot to *value*.

`symbol-pref` *symbol* [Scheme Procedure]
`scm_symbol_pref` (*symbol*) [C Function]
Return the *property list* currently associated with *symbol*.

`symbol-pset!` *symbol value* [Scheme Procedure]
`scm_symbol_pset_x` (*symbol, value*) [C Function]
Set *symbol*'s property list to *value*.

`symbol-property` *sym prop* [Scheme Procedure]
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* [Scheme Procedure]
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* [Scheme Procedure]

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 267.

6.6.7.6 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.

6.6.7.7 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.

`make-symbol` *name* [Scheme Procedure]
`scm_make_symbol` (*name*) [C Function]

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.

`symbol-interned?` *symbol* [Scheme Procedure]
`scm_symbol_interned_p` (*symbol*) [C Function]

Return `#t` if *symbol* is interned, otherwise return `#f`.

For example:

```
(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.
```

6.6.8 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 `:`.

6.6.8.1 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.

6.6.8.2 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 240.

6.6.8.3 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 6.6.7.6 [Symbol Read Syntax], page 176), 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 `keyword` 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 `keyword` read option is set to `'postfix`, Guile recognizes the SRFI-88 read syntax `NAME:` (see Section 7.5.34 [SRFI-88], page 590). 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 6.17.2 [Scheme Read], page 346. Note that the `prefix` and `postfix` syntax are mutually exclusive.

```
(read-set! keywords 'prefix)

#:type
⇒
#:type

:type
⇒
#:type

(read-set! keywords 'postfix)

type:
⇒
#:type

:type
⇒
:type

(read-set! keywords #f)

#:type
⇒
#:type

:type
```

```
⊣
ERROR: In expression :type:
ERROR: Unbound variable: :type
ABORT: (unbound-variable)
```

6.6.8.4 Keyword Procedures

`keyword?` *obj* [Scheme Procedure]
`scm_keyword_p` (*obj*) [C Function]

Return `#t` if the argument *obj* is a keyword, else `#f`.

`keyword->symbol` *keyword* [Scheme Procedure]
`scm_keyword_to_symbol` (*keyword*) [C Function]

Return the symbol with the same name as *keyword*.

`symbol->keyword` *symbol* [Scheme Procedure]
`scm_symbol_to_keyword` (*symbol*) [C Function]

Return the keyword with the same name as *symbol*.

`int scm_is_keyword` (*SCM obj*) [C Function]

Equivalent to `scm_is_true (scm_keyword_p (obj))`.

`SCM scm_from_locale_keyword` (*const char *name*) [C Function]
`SCM scm_from_locale_keywordn` (*const char *name, size_t len*) [C Function]

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 source code. In such cases, use `scm_from_latin1_keyword` or `scm_from_utf8_keyword`.

`SCM scm_from_latin1_keyword` (*const char *name*) [C Function]
`SCM scm_from_utf8_keyword` (*const char *name*) [C Function]

Equivalent to `scm_symbol_to_keyword (scm_from_latin1_symbol (name))` and `scm_symbol_to_keyword (scm_from_utf8_symbol (name))`, respectively.

6.6.9 "Functionality-Centric" Data Types

Procedures and macros are documented in their own sections: see Section 6.9 [Procedures], page 236 and Section 6.10 [Macros], page 248.

Variable objects are documented as part of the description of Guile's module system: see Section 6.19.7 [Variables], page 373.

Asyncs, dynamic roots and fluids are described in the section on scheduling: see Section 6.21 [Scheduling], page 396.

Hooks are documented in the section on general utility functions: see Section 6.11.6 [Hooks], page 270.

Ports are described in the section on I/O: see Section 6.14 [Input and Output], page 304.

Regular expressions are described in their own section: see Section 6.15 [Regular Expressions], page 335.

6.7 Compound Data Types

This chapter describes Guile's compound data types. By *compound* we mean that the primary purpose of these data types is to act as containers for other kinds of data (including other compound objects). For instance, a (non-uniform) vector with length 5 is a container that can hold five arbitrary Scheme objects.

The various kinds of container object differ from each other in how their memory is allocated, how they are indexed, and how particular values can be looked up within them.

6.7.1 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 6.7.2 [Lists], page 185).

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 6.17.1.1 [Expression Syntax], page 343. 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.

`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.

`cddr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]
`caddr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]
`caaar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]
`cdddar` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]

`caaaar` *pair* [Scheme Procedure]
`scm_cddr` (*pair*) [C Function]
`scm_cdar` (*pair*) [C Function]
`scm_cadr` (*pair*) [C Function]
`scm_caar` (*pair*) [C Function]
`scm_cdddr` (*pair*) [C Function]
`scm_cddar` (*pair*) [C Function]
`scm_cdadr` (*pair*) [C Function]
`scm_cdaar` (*pair*) [C Function]
`scm_caddr` (*pair*) [C Function]
`scm_cadar` (*pair*) [C Function]
`scm_caadr` (*pair*) [C Function]
`scm_caaar` (*pair*) [C Function]
`scm_cddddr` (*pair*) [C Function]
`scm_cdddar` (*pair*) [C Function]
`scm_cddadr` (*pair*) [C Function]
`scm_cddaar` (*pair*) [C Function]
`scm_cdaddr` (*pair*) [C Function]
`scm_cdadar` (*pair*) [C Function]
`scm_cdaadr` (*pair*) [C Function]
`scm_cdaaar` (*pair*) [C Function]
`scm_cadddr` (*pair*) [C Function]
`scm_caddar` (*pair*) [C Function]
`scm_cadadr` (*pair*) [C Function]
`scm_cadaar` (*pair*) [C Function]
`scm_caaddr` (*pair*) [C Function]
`scm_caadar` (*pair*) [C Function]
`scm_caaadr` (*pair*) [C Function]
`scm_caaaar` (*pair*) [C Function]

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

`cadr`, `caddr` 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 536).

`set-car!` *pair value* [Scheme Procedure]
`scm_set_car_x` (*pair, value*) [C Function]

Stores *value* in the car field of *pair*. The value returned by `set-car!` is unspecified.

`set-cdr!` *pair value* [Scheme Procedure]
`scm_set_cdr_x` (*pair, value*) [C Function]

Stores *value* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

6.7.2 Lists

A very important data type in Scheme—as well as in all other Lisp dialects—is the data type *list*.[2]

This is the short definition of what a list is:

- Either the empty list `()`,
- or a pair which has a list in its cdr.

6.7.2.1 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.[3].

```
(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 6.17.1.1 [Expression Syntax], page 343), because they would otherwise be mistakingly taken as procedure applications (see Section 3.2.2 [Simple Invocation], page 18).

6.7.2.2 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.

`list?` *x* [Scheme Procedure]
`scm_list_p` (*x*) [C Function]

Return `#t` iff *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* [Scheme Procedure]
`scm_null_p` (*x*) [C Function]

Return `#t` iff *x* is the empty list, else `#f`.

---

[2] 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.

[3] Note that there is no separation character between the list elements, like a comma or a semicolon.

`int scm_is_null` (*SCM x*) [C Function]

Return 1 when *x* is the empty list; otherwise return 0.

6.7.2.3 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 ...* [Scheme Procedure]
`scm_list_1` (*elem1*) [C Function]
`scm_list_2` (*elem1, elem2*) [C Function]
`scm_list_3` (*elem1, elem2, elem3*) [C Function]
`scm_list_4` (*elem1, elem2, elem3, elem4*) [C Function]
`scm_list_5` (*elem1, elem2, elem3, elem4, elem5*) [C Function]
`scm_list_n` (*elem1, ..., elemN,* `SCM_UNDEFINED`) [C Function]

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 ...* [Scheme Procedure]

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* [Scheme Procedure]
`scm_list_copy` (*lst*) [C Function]

Return a (newly-created) copy of *lst*.

`make-list` *n* [*init*] [Scheme Procedure]

Create a list containing of *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 270).

6.7.2.4 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]
`scm_length` (*lst*) [C Function]

Return the number of elements in list *lst*.

`last-pair` *lst* [Scheme Procedure]
`scm_last_pair` (*lst*) [C Function]

Return the last pair in *lst*, signalling an error if *lst* is circular.

`list-ref` *list k* [Scheme Procedure]
`scm_list_ref` (*list*, *k*) [C Function]

Return the *k*th element from *list*.

`list-tail` *lst k* [Scheme Procedure]
`list-cdr-ref` *lst k* [Scheme Procedure]
`scm_list_tail` (*lst*, *k*) [C Function]

Return the "tail" of *lst* beginning with its *k*th element. The first element of the list is considered to be element 0.

`list-tail` and `list-cdr-ref` are identical. It may help to think of `list-cdr-ref` as accessing the *k*th cdr of the list, or returning the results of cdring *k* times down *lst*.

`list-head` *lst k* [Scheme Procedure]
`scm_list_head` (*lst*, *k*) [C Function]

Copy the first *k* elements from *lst* into a new list, and return it.

6.7.2.5 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` [Scheme Procedure]
`append!` *lst* ... *obj* [Scheme Procedure]
`append!` [Scheme Procedure]
`scm_append` (*lstlst*) [C Function]
`scm_append_x` (*lstlst*) [C Function]

Return a list comprising all the elements of lists *lst* ... *obj*. If called with no arguments, return the empty list.

```
(append '(x) '(y))          ⇒  (x y)
(append '(a) '(b c d))      ⇒  (a b c d)
(append '(a (b)) '((c)))    ⇒  (a (b) (c))
```

The last argument *obj* may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))    ⇒  (a b c . d)
(append '() 'a)             ⇒  a
```

`append` doesn't modify the given lists, but the return may share structure with the final *obj*. `append!` modifies the given lists to form its return.

For `scm_append` and `scm_append_x`, *lstlst* is a list of the list operands *lst* ... *obj*. That *lstlst* itself is not modified or used in the return.

`reverse` *lst* [Scheme Procedure]
`reverse!` *lst* [*newtail*] [Scheme Procedure]
`scm_reverse` (*lst*) [C Function]
`scm_reverse_x` (*lst*, *newtail*) [C Function]

Return a list comprising the elements of *lst*, in reverse order.

`reverse` constructs a new list, `reverse!` modifies *lst* in constructing its return.

For `reverse!`, the optional *newtail* is appended to the result. *newtail* isn't reversed, it simply becomes the list tail. For `scm_reverse_x`, the *newtail* parameter is mandatory, but can be `SCM_EOL` if no further tail is required.

6.7.2.6 List Modification

The following procedures modify an existing list, either by changing elements of the list, or by changing the list structure itself.

`list-set!` *list k val* [Scheme Procedure]
`scm_list_set_x` (*list*, *k*, *val*) [C Function]

Set the *k*th element of *list* to *val*.

`list-cdr-set!` *list k val* [Scheme Procedure]
`scm_list_cdr_set_x` (*list*, *k*, *val*) [C Function]

Set the *k*th cdr of *list* to *val*.

`delq` *item lst* [Scheme Procedure]
`scm_delq` (*item*, *lst*) [C Function]

Return a newly-created copy of *lst* with elements `eq?` to *item* removed. This procedure mirrors `memq`: `delq` compares elements of *lst* against *item* with `eq?`.

`delv` *item lst* [Scheme Procedure]
`scm_delv` (*item*, *lst*) [C Function]

Return a newly-created copy of *lst* with elements `eqv?` to *item* removed. This procedure mirrors `memv`: `delv` compares elements of *lst* against *item* with `eqv?`.

`delete` *item lst* [Scheme Procedure]
`scm_delete` (*item*, *lst*) [C Function]

Return a newly-created copy of *lst* with elements `equal?` to *item* removed. This procedure mirrors `member`: `delete` compares elements of *lst* against *item* with `equal?`.

See also SRFI-1 which has an extended `delete` (Section 7.5.3.8 [SRFI-1 Deleting], page 544), and also an `lset-difference` which can delete multiple *items* in one call (Section 7.5.3.10 [SRFI-1 Set Operations], page 545).

`delq!` *item lst* [Scheme Procedure]
`delv!` *item lst* [Scheme Procedure]
`delete!` *item lst* [Scheme Procedure]
`scm_delq_x` (*item*, *lst*) [C Function]
`scm_delv_x` (*item*, *lst*) [C Function]
`scm_delete_x` (*item*, *lst*) [C Function]

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.

`delq1!` *item lst* [Scheme Procedure]
`scm_delq1_x` (*item, lst*) [C Function]
Like `delq!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eq?`. See also `delv1!` and `delete1!`.

`delv1!` *item lst* [Scheme Procedure]
`scm_delv1_x` (*item, lst*) [C Function]
Like `delv!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eqv?`. See also `delq1!` and `delete1!`.

`delete1!` *item lst* [Scheme Procedure]
`scm_delete1_x` (*item, lst*) [C Function]
Like `delete!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `equal?`. See also `delq1!` and `delv1!`.

`filter` *pred lst* [Scheme Procedure]
`filter!` *pred lst* [Scheme Procedure]
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.

6.7.2.7 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.

`memq` *x lst* [Scheme Procedure]
`scm_memq` (*x, lst*) [C Function]
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* [Scheme Procedure]
`scm_memv` (*x, lst*) [C Function]
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* [Scheme Procedure]
`scm_member` (*x, lst*) [C Function]
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 542).

6.7.2.8 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 . . .* [Scheme Procedure]
`map-in-order` *proc arg1 arg2 . . .* [Scheme Procedure]
`scm_map` (*proc, arg1, args*) [C Function]

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 . . .* [Scheme Procedure]

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 538).

6.7.3 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 6.7.6 [Arrays], page 198).

6.7.3.1 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. 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)
```

Like lists, vectors have to be quoted:

```
'#(a b c) ⇒ #(a b c)
```

6.7.3.2 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* [Scheme Procedure]
`scm_vector_p` (*obj*) [C Function]

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`.

6.7.3.3 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]
`scm_vector_length` (*vector*) [C Function]

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]
`scm_vector_ref` (*vec, k*) [C Function]

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 6.7.3.2 [Vector Creation], page 191) 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]
`scm_vector_set_x` (*vec, k, obj*) [C Function]

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* [Scheme Procedure]
`scm_vector_fill_x` (*vec, fill*) [C Function]

Store *fill* in every position of *vec*. The value returned by `vector-fill!` is unspecified.

`vector-copy` *vec* [Scheme Procedure]
`scm_vector_copy` (*vec*) [C Function]

Return a copy of *vec*.

`vector-move-left!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_left_x` (*vec1, start1, end1, vec2, start2*) [C Function]

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* [Scheme Procedure]
`scm_vector_move_right_x` (*vec1, start1, end1, vec2, start2*) [C Function]

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*.

6.7.3.4 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 6.7.6.4 [Accessing Arrays from C], page 205), 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*) [C Function]

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*) [C Macro]

Evaluates to the length of the simple vector *vec*. No type checking is done.

`SCM SCM_SIMPLE_VECTOR_REF` (*SCM vec, size_t idx*) [C Macro]

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*) [C Macro]

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, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

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 6.7.6.3 [Shared Arrays], page 203).

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, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

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);
```

6.7.3.5 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 549, 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 6.6.6 [Bytevectors], page 162, for more information on bytevectors.

6.7.4 Bit Vectors

Bit vectors are zero-origin, one-dimensional arrays of booleans. They are displayed as a sequence of `0`s and `1`s prefixed by `#*`, e.g.,

```
(make-bitvector 8 #f) ⇒
#*00000000
```

Bit vectors are also generalized vectors, See Section 6.7.5 [Generalized Vectors], page 197, and can thus be used with the array procedures, See Section 6.7.6 [Arrays], page 198. Bit vectors are the special case of one dimensional bit arrays.

`bitvector?` *obj* [Scheme Procedure]
`scm_bitvector_p` (*obj*) [C Function]
Return `#t` when *obj* is a bitvector, else return `#f`.

`int scm_is_bitvector` (*SCM obj*) [C Function]
Return `1` when *obj* is a bitvector, else return `0`.

`make-bitvector` *len* [*fill*] [Scheme Procedure]
`scm_make_bitvector` (*len, fill*) [C Function]
Create a new bitvector of length *len* and optionally initialize all elements to *fill*.

`SCM scm_c_make_bitvector` (*size_t len, SCM fill*) [C Function]
Like `scm_make_bitvector`, but the length is given as a `size_t`.

`bitvector` *bit* ... [Scheme Procedure]
`scm_bitvector` (*bits*) [C Function]
Create a new bitvector with the arguments as elements.

`bitvector-length` *vec* [Scheme Procedure]
`scm_bitvector_length` (*vec*) [C Function]
Return the length of the bitvector *vec*.

`size_t scm_c_bitvector_length` (*SCM vec*) [C Function]
Like `scm_bitvector_length`, but the length is returned as a `size_t`.

`bitvector-ref` *vec idx* [Scheme Procedure]
`scm_bitvector_ref` (*vec, idx*) [C Function]
Return the element at index *idx* of the bitvector *vec*.

`SCM scm_c_bitvector_ref` (*SCM vec, size_t idx*) [C Function]
Return the element at index *idx* of the bitvector *vec*.

`bitvector-set!` *vec idx val* [Scheme Procedure]
`scm_bitvector_set_x` (*vec, idx, val*) [C Function]
Set the element at index *idx* of the bitvector *vec* when *val* is true, else clear it.

`SCM scm_c_bitvector_set_x` (*SCM vec, size_t idx, SCM val*) [C Function]
Set the element at index *idx* of the bitvector *vec* when *val* is true, else clear it.

`bitvector-fill!` *vec val* [Scheme Procedure]
`scm_bitvector_fill_x` (*vec, val*) [C Function]
Set all elements of the bitvector *vec* when *val* is true, else clear them.

`list->bitvector` *list* [Scheme Procedure]
`scm_list_to_bitvector` (*list*) [C Function]
Return a new bitvector initialized with the elements of *list*.

`bitvector->list` *vec* [Scheme Procedure]
`scm_bitvector_to_list` (*vec*) [C Function]
Return a new list initialized with the elements of the bitvector *vec*.

`bit-count` *bool bitvector* [Scheme Procedure]
`scm_bit_count` (*bool, bitvector*) [C Function]

Return a count of how many entries in *bitvector* are equal to *bool*. For example,

```
(bit-count #f #*000111000)  ⇒ 6
```

`bit-position` *bool bitvector start* [Scheme Procedure]
`scm_bit_position` (*bool, bitvector, start*) [C Function]

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* [Scheme Procedure]
`scm_bit_invert_x` (*bitvector*) [C Function]

Modify *bitvector* by replacing each element with its negation.

`bit-set*!` *bitvector uvec bool* [Scheme Procedure]
`scm_bit_set_star_x` (*bitvector, uvec, bool*) [C Function]

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
⇒ #*11010011
```

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
```

`bit-count*` *bitvector uvec bool* [Scheme Procedure]
`scm_bit_count_star` (*bitvector, uvec, bool*) [C Function]

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 #u(7 0 4) #f)  ⇒ 2
```

`const scm_t_uint32 * scm_bitvector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` (see Section 6.7.3.4 [Vector Accessing from C], page 193), 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.

`scm_t_uint32 * scm_bitvector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_bitvector_elements`, but the pointer is good for reading and writing.

6.7.5 Generalized Vectors

Guile has a number of data types that are generally vector-like: strings, uniform numeric vectors, bytevectors, bitvectors, and of course ordinary vectors of arbitrary Scheme values. These types are disjoint: a Scheme value belongs to at most one of the five types listed above.

If you want to gloss over this distinction and want to treat all four types with common code, you can use the procedures in this section. They work with the *generalized vector* type, which is the union of the five vector-like types.

`generalized-vector?` *obj* [Scheme Procedure]
`scm_generalized_vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a vector, bytevector, string, bitvector, or uniform numeric vector.

`generalized-vector-length` *v* [Scheme Procedure]
`scm_generalized_vector_length` (*v*) [C Function]

Return the length of the generalized vector *v*.

`generalized-vector-ref` *v idx* [Scheme Procedure]
`scm_generalized_vector_ref` (*v, idx*) [C Function]

Return the element at index *idx* of the generalized vector *v*.

`generalized-vector-set!` *v idx val* [Scheme Procedure]
`scm_generalized_vector_set_x` (*v, idx, val*) [C Function]

Set the element at index *idx* of the generalized vector *v* to *val*.

`generalized-vector->list` *v* [Scheme Procedure]
`scm_generalized_vector_to_list` (*v*) [C Function]

Return a new list whose elements are the elements of the generalized vector *v*.

`int scm_is_generalized_vector` (*SCM obj*) [C Function]

Return `1` if *obj* is a vector, string, bitvector, or uniform numeric vector; else return `0`.

`size_t scm_c_generalized_vector_length` (*SCM v*) [C Function]

Return the length of the generalized vector *v*.

`SCM scm_c_generalized_vector_ref` (*SCM v, size_t idx*) [C Function]

Return the element at index *idx* of the generalized vector *v*.

`void scm_c_generalized_vector_set_x` (*SCM v, size_t idx, SCM val*) [C Function]

Set the element at index *idx* of the generalized vector *v* to *val*.

`void scm_generalized_vector_get_handle` (*SCM v, scm_t_array_handle *handle*) [C Function]

Like `scm_array_get_handle` but an error is signalled when *v* is not of rank one. You can use `scm_array_handle_ref` and `scm_array_handle_set` to read and write the elements of *v*, or you can use functions like `scm_array_handle_<foo>_elements` to deal with specific types of vectors.

6.7.6 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 generalized vector for the actual storage of its elements. Any kind of generalized 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.

Generalized vectors, such as strings, uniform numeric vectors, bytevectors, bit vectors and ordinary vectors, are the special case of one dimensional arrays.

6.7.6.1 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 3×3 matrix with index ranges 0..2 and 0..2.

`#u32(0 1 2)`
is a uniform u8 array of rank 1.

`#2u32@2@3((1 2) (2 3))`
is a uniform u8 array of rank 2 with index ranges 2..3 and 3..4.

`#2()` is a two-dimensional array with index ranges 0..-1 and 0..-1, i.e. both dimensions have length zero.

`#2:0:2()` 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.

`#0(12)` is a rank-zero array with contents 12.

In addition, bytevectors are also arrays, but use a different syntax (see Section 6.6.6 [Bytevectors], page 162):

`#vu8(1 2 3)`
is a 3-byte long bytevector, with contents 1, 2, 3.

6.7.6.2 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*) [C Function]

Create and return an array that has as many dimensions as there are *bound*s 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]

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]

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 index 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* ... [Scheme Procedure]
`scm_array_set_x` (*array, obj, idxlist*) [C Function]

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* [Scheme Procedure]
`array-dimensions` *array* [Scheme Procedure]
`scm_array_dimensions` (*array*) [C Function]

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-rank` *array* [Scheme Procedure]
`scm_array_rank` (*array*) [C Function]

Return the rank of *array*.

`size_t scm_c_array_rank` (*SCM array*) [C Function]

Return the rank of *array* as a `size_t`.

`array->list` *array* [Scheme Procedure]
`scm_array_to_list` (*array*) [C Function]

Return a list consisting of all the elements, in order, of *array*.

`array-copy!` *src dst* [Scheme Procedure]
`array-copy-in-order!` *src dst* [Scheme Procedure]
`scm_array_copy_x` (*src, dst*) [C Function]

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* [Scheme Procedure]
`scm_array_fill_x` (*array, fill*) [C Function]

Store *fill* in every element of *array*. The value returned is unspecified.

`array-equal?` *array* ... [Scheme Procedure]

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 265) in that all arguments must be arrays.

`array-map!` *dst proc src* ... [Scheme Procedure]
`array-map-in-order!` *dst proc src1* ... *srcN* [Scheme Procedure]
`scm_array_map_x` (*dst, proc, srclist*) [C Function]

Set each element of the *dst* array to values obtained from calls to *proc*. The value returned is unspecified.

Each call is `(proc elem1 ... elemN)`, 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* ... [Scheme Procedure]
`scm_array_for_each` (*proc, src1, srclist*) [C Function]

Apply *proc* to each tuple of elements of *src1 src2* ..., in row-major order. The value returned is unspecified.

`array-index-map!` *dst proc* [Scheme Procedure]
`scm_array_index_map_x` (*dst, proc*) [C Function]

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)))
```

`uniform-array-read!` *ra* [*port_or_fd* [*start* [*end*]]] [Scheme Procedure]
`scm_uniform_array_read_x` (*ra, port_or_fd, start, end*) [C Function]

Attempt to read all elements of array *ra*, in lexicographic order, as binary objects from *port_or_fd*. If an end of file is encountered, the objects up to that point are put into *ra* (starting at the beginning) and the remainder of the array is unchanged.

The optional arguments *start* and *end* allow a specified region of a vector (or linearized array) to be read, leaving the remainder of the vector unchanged.

`uniform-array-read!` returns the number of objects read. *port_or_fd* may be omitted, in which case it defaults to the value returned by `(current-input-port)`.

`uniform-array-write` *ra* [*port_or_fd* [*start* [*end*]]] [Scheme Procedure]
`scm_uniform_array_write` (*ra, port_or_fd, start, end*) [C Function]

Writes all elements of *ra* as binary objects to *port_or_fd*.

The optional arguments *start* and *end* allow a specified region of a vector (or linearized array) to be written.

The number of objects actually written is returned. *port_or_fd* may be omitted, in which case it defaults to the value returned by `(current-output-port)`.

6.7.6.3 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 6.7.6.2 [Array Procedures], page 199).

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)))
                   3 3)
⇒ #2((c b a) (f e d) (i h g))
```

A fixed offset on indexes allows for instance a change from a 0 based to a 1 based array,

```
(define x #2((a b c) (d e f) (g h i)))
(define y (make-shared-array x
                             (lambda (i j) (list (1- i) (1- j)))
                             '(1 3) '(1 3)))
(array-ref x 0 0) ⇒ a
(array-ref y 1 1) ⇒ a
```

A multiple on an index allows every Nth element of an array to be taken. The following is every third element,

```
(make-shared-array #1(a b c d e f g h i j k l)
                   (lambda (i) (list (* i 3)))
                   4)
⇒ #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* [Scheme Procedure]
`scm_shared_array_increments` (*array*) [C Function]

For each dimension, return the distance between elements in the root vector.

`shared-array-offset` *array* [Scheme Procedure]
`scm_shared_array_offset` (*array*) [C Function]

Return the root vector index of the first element in the array.

`shared-array-root` *array* [Scheme Procedure]
`scm_shared_array_root` (*array*) [C Function]

Return the root vector of a shared array.

`array-contents` *array* [*strict*] [Scheme Procedure]
`scm_array_contents` (*array, strict*) [C Function]

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* ... [Scheme Procedure]
`scm_transpose_array` (*array, dimlist*) [C Function]

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*, *dim2*, ... 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 *dim1*, *dim2*, ... correspond to dimensions in the array to be returned, and their positions in the argument list to dimensions of *array*. Several *dim*s may have the same value, in which case the returned array will have smaller rank than *array*.

```
(transpose-array '#2((a b) (c d)) 1 0) ⇒ #2((a c) (b d))
(transpose-array '#2((a b) (c d)) 0 0) ⇒ #1(a d)
(transpose-array '#3(((a b c) (d e f)) ((1 2 3) (4 5 6))) 1 1 0) ⇒
                #2((a 4) (b 5) (c 6))
```

6.7.6.4 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 298).

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] + ... + 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 3x3 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` [C Type]

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*) [C Function]

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*) [C Function]

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*) [C Function]

Return the rank of the array represented by *handle*.

`scm_t_array_dim` [C Type]

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` (*scm_t_array_handle *handle*) [C Function]

Return 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 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*:

```
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` (*scm_t_array_handle *handle, SCM indices*) [C Function]

Compute 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` (*scm_t_array_handle *handle, ssize_t pos*) [C Function]

Return 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]

`const double * scm_array_handle_f64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_c32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const double * scm_array_handle_c64_elements` [C Function]
(*scm_t_array_handle *handle*)

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.

`scm_t_uint8 * scm_array_handle_u8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int8 * scm_array_handle_s8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint16 * scm_array_handle_u16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int16 * scm_array_handle_s16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint32 * scm_array_handle_u32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int32 * scm_array_handle_s32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint64 * scm_array_handle_u64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int64 * scm_array_handle_s64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`float * scm_array_handle_f32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`double * scm_array_handle_f64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`float * scm_array_handle_c32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`double * scm_array_handle_c64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_<kind>_elements`, but the pointer is good for reading and writing.

`const scm_t_uint32 * scm_array_handle_bit_elements` [C Function]
(*scm_t_array_handle *handle*)

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*:

```
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 = ...
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);
```

`scm_t_uint32 * scm_array_handle_bit_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_bit_elements` but the pointer is good for reading and writing. You must take care not to modify bits outside of the allowed index range of the array, even for contiguous arrays.

6.7.7 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 `block-growth-factor` (2 by default).

The VList structure also serves as the basis for the *VList-based hash lists* or "vhashes", an immutable dictionary type (see Section 6.7.12 [VHashes], page 226).

However, the current implementation in `(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 `vlist-cons` mutates part of its internal structure, which makes it non-thread-safe. This could be fixed, but it would slow down `vlist-cons`.
- `vlist-cons` always allocates at least as much memory as `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.
- `vlist-cons` is a Scheme procedure compiled to bytecode, and it does not compete with the straightforward C implementation of `cons`, and with the fact that the VM has a special `cons` instruction.

We hope to address these in the future.

The programming interface exported by `(ice-9 vlist)` is defined below. Most of it is the same as SRFI-1 with an added `vlist-` prefix to function names.

`vlist?` *obj* [Scheme Procedure]
Return true if *obj* is a VList.

`vlist-null` [Scheme Variable]
The empty VList. Note that it's possible to create an empty VList not `eq?` to `vlist-null`; thus, callers should always use `vlist-null?` when testing whether a VList is empty.

`vlist-null?` *vlist* [Scheme Procedure]
Return true if *vlist* is empty.

`vlist-cons` *item vlist* [Scheme Procedure]
Return a new vlist with *item* as its head and *vlist* as its tail.

`vlist-head` *vlist* [Scheme Procedure]
Return the head of *vlist*.

`vlist-tail` *vlist* [Scheme Procedure]
Return the tail of *vlist*.

`block-growth-factor` [Scheme Variable]
A fluid that defines the growth factor of VList blocks, 2 by default.

The functions below provide the usual set of higher-level list operations.

`vlist-fold` *proc init vlist* [Scheme Procedure]
`vlist-fold-right` *proc init vlist* [Scheme Procedure]
Fold over *vlist*, calling *proc* for each element, as for SRFI-1 `fold` and `fold-right` (see Section 7.5.3 [SRFI-1], page 535).

`vlist-ref` *vlist index* [Scheme Procedure]
Return the element at index *index* in *vlist*. This is typically a constant-time operation.

`vlist-length` *vlist* [Scheme Procedure]
Return the length of *vlist*. This is typically logarithmic in the number of elements in *vlist*.

`vlist-reverse` *vlist* [Scheme Procedure]
Return a new *vlist* whose content are those of *vlist* in reverse order.

`vlist-map` *proc vlist* [Scheme Procedure]
Map *proc* over the elements of *vlist* and return a new vlist.

`vlist-for-each` *proc vlist* [Scheme Procedure]
Call *proc* on each element of *vlist*. The result is unspecified.

`vlist-drop` *vlist count* [Scheme Procedure]
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* [Scheme Procedure]
Return a new vlist that contains only the *count* first elements of *vlist*.

`vlist-filter` *pred vlist* [Scheme Procedure]
Return a new vlist containing all the elements from *vlist* that satisfy *pred*.

`vlist-delete` *x vlist* [*equal?*] [Scheme Procedure]
Return a new vlist corresponding to *vlist* without the elements *equal?* to *x*.

`vlist-unfold` *p f g seed* [*tail-gen*] [Scheme Procedure]
`vlist-unfold-right` *p f g seed* [*tail*] [Scheme Procedure]
Return a new vlist, as for SRFI-1 `unfold` and `unfold-right` (see Section 7.5.3 [SRFI-1], page 535).

`vlist-append` *vlist* ... [Scheme Procedure]
Append the given vlists and return the resulting vlist.

`list->vlist` *lst* [Scheme Procedure]
Return a new vlist whose contents correspond to *lst*.

`vlist->list` *vlist* [Scheme Procedure]
Return a new list whose contents match those of *vlist*.

6.7.8 Records

A *record type* is a first class object representing a user-defined data type. A *record* is an instance of a record type.

`record?` *obj* [Scheme Procedure]
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*] [Scheme Procedure]

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*] [Scheme Procedure]

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 `make-record-type` that created the type represented by *rtd*; if the *field-names* argument is provided, it is an error if it contains any duplicates or any symbols not in the default list.

`record-predicate` *rtd* [Scheme Procedure]

Return a procedure for testing membership in the type represented by *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.

`record-accessor` *rtd field-name* [Scheme Procedure]

Return a procedure for reading the value of a particular field of a member of the type represented by *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 *field-name* in that record. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

`record-modifier` *rtd field-name* [Scheme Procedure]

Return a procedure for writing the value of a particular field of a member of the type represented by *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 *field-name* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field-name* must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-descriptor` *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 `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 `record-constructor` in the call that created the constructor procedure that created the given record.

`record-type-name` *rtd* [Scheme Procedure]

Return the type-name associated with the type represented by rtd. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-fields` *rtd* [Scheme Procedure]

Return a list of the symbols naming the fields in members of the type represented by *rtd*. The returned value is `equal?` to the field-names argument given in the call to `make-record-type` that created the type represented by *rtd*.

6.7.9 Structures

A *structure* is a first class data type which holds Scheme values or C words in fields numbered 0 upwards. A *vtable* represents a structure type, giving field types and permissions, and an optional print function for `write` etc.

Structures are lower level than records (see Section 6.7.8 [Records], page 212) but have some extra features. The vtable system allows sets of types be constructed, with class data. The uninterpreted words can inter-operate with C code, allowing arbitrary pointers or other values to be stored along side usual Scheme `SCM` values.

6.7.9.1 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 6.7.9.3 [Vtable Contents], page 216.)

`make-vtable` *fields* [*print*] [Scheme Procedure]

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 "uninterpreted" (it's not treated as a Scheme value), or "unprotected" (it's not marked during GC), or "unsigned long" (its size), or all of these things.
- `s` – a self-reference. Such a field holds the `SCM` value of the structure itself (a circular reference). This can be useful in C code where you might have a pointer to the data array, and want to get the Scheme `SCM` handle for the structure. In Scheme code it has no use.

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.
- `o` – opaque, the field can be neither read nor written at the Scheme level. This can be used for fields which should only be used from C code.
- `W`,`R`,`O` – a tail array, with permissions for the array fields as per `w`,`r`,`o`.

A tail array is further fields at the end of a structure. The last field in the layout string might be for instance 'pW' to have a tail of writable Scheme-valued fields. The 'pW' field itself holds the tail size, and the tail fields come after it.

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 uninterpreted

(make-vtable "prpW")    ;; one fixed then a tail array
```

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)
               (display "#<" port)
               (display (struct-ref struct 0) port)
               (display " and " port)
               (display (struct-ref struct 1) port)
               (display ">" port)))
```

6.7.9.2 Structure Basics

This section describes the basic procedures for working with structures. `make-struct` creates a structure, and `struct-ref` and `struct-set!` access write fields.

`make-struct` *vtable tail-size init . . .* [Scheme Procedure]
`scm_make_struct` (*vtable, tail_size, init_list*) [C Function]

Create a new structure, with layout per the given *vtable* (see Section 6.7.9.1 [Vtables], page 214).

tail-size is the size of the tail array if *vtable* specifies a tail array. *tail-size* should be 0 when *vtable* doesn't specify a tail array.

The optional *init. . .* arguments are initial values for the fields of the structure (and the tail array). 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 uninterpreted field (type `u`).

Type `s` self-reference fields, permission `o` opaque fields, and the count field of a tail array are all ignored for the *init* arguments, ie. an argument is not consumed by such a field. An `s` is always set to the structure itself, an `o` is always set to `#f` or 0 (with the intention that C code will do something to it later), and the tail count is always the given *tail-size*.

For example,

```
(define v (make-vtable "prpwpw"))
(define s (make-struct v 0 123 "abc" 456))
(struct-ref s 0) ⇒ 123
```

```
(struct-ref s 1) ⇒ "abc"
(define v (make-vtable "prpW"))
(define s (make-struct v 6 "fixed field" 'x 'y))
(struct-ref s 0) ⇒ "fixed field"
(struct-ref s 1) ⇒ 2    ;; tail size
(struct-ref s 2) ⇒ x    ;; tail array ...
(struct-ref s 3) ⇒ y
(struct-ref s 4) ⇒ #f
```

`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, or if the field cannot be read because it's `o` opaque.

`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 `r` read-only or `o` opaque.

`struct-vtable` *struct* [Scheme Procedure]
`scm_struct_vtable` (*struct*) [C Function]

Return the vtable used by *struct*.

This can be used to examine the layout of an unknown structure, see Section 6.7.9.3 [Vtable Contents], page 216.

6.7.9.3 Vtable Contents

A vtable is itself a structure, with particular fields that hold information about the structures to be created. These include the fields of those structures, and the print function for them. The variables below allow access to those fields.

`struct-vtable?` *obj* [Scheme Procedure]
`scm_struct_vtable_p` (*obj*) [C Function]

Return `#t` if *obj* is a vtable structure.

Note that because vtables are simply structures with a particular layout, `struct-vtable?` can potentially return true on an application structure which merely happens to look like a vtable.

`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 6.7.9.4 [Vtable Vtables], page 217).

```
(define v (make-vtable "pwpw" 0))
(struct-ref v vtable-index-layout) ⇒ pwpw
```

This field is read-only, since the layout of structures using a vtable cannot be changed.

`vtable-index-vtable` [Scheme Variable]
`scm_vtable_index_vtable` [C Macro]

A self-reference to the vtable, ie. a type `s` field. This is used by C code within Guile and has no use at the Scheme level.

`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.

```
(define (my-print-func struct port)
  ...)
(define v (make-vtable "pwpw" my-print-func))
(struct-ref v vtable-index-printer) ⇒ 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*.

```
(define v (make-vtable "pw"))
(set-struct-vtable-name! v 'my-name)

(define s (make-struct v 0))
(display s) ⊣ #<my-name b7ab3ae0:b7ab3730>
```

6.7.9.4 Vtable Vtables

As noted above, a vtable is a structure and that structure is itself described by a vtable. Such a "vtable of a vtable" can be created with `make-vtable-vtable` below. This can be used to build sets of related vtables, possibly with extra application fields.

This second level of vtable can be a little confusing. The ball example below is a typical use, adding a "class data" field to the vtables, from which instance structures are created. The current implementation of Guile's own records (see Section 6.7.8 [Records], page 212) does something similar, a record type descriptor is a vtable with room to hold the field names of the records to be created from it.

`make-vtable-vtable` *user-fields tail-size* [*print*] [Scheme Procedure]
`scm_make_vtable_vtable` (*user_fields, tail_size, print_and_init_list*) [C Function]

Create a "vtable-vtable" which can be used to create vtables. This vtable-vtable is also a vtable, and is self-describing, meaning its vtable is itself. The following is a simple usage.

```
(define vt-vt (make-vtable-vtable "" 0))
(define vt    (make-struct vt-vt 0
                           (make-struct-layout "pwpw"))
(define s     (make-struct vt 0 123 456))

(struct-ref s 0) ⇒ 123
```

`make-struct` is used to create a vtable from the vtable-vtable. The first initializer is a layout object (field `vtable-index-layout`), usually obtained from `make-struct-layout` (below). An optional second initializer is a printer function (field `vtable-index-printer`), used as described under `make-vtable` (see Section 6.7.9.1 [Vtables], page 214).

user-fields is a layout string giving extra fields to have in the vtables. A vtable starts with some base fields as per Section 6.7.9.3 [Vtable Contents], page 216, and *user-fields* is appended. The *user-fields* start at field number `vtable-offset-user` (below), and exist in both the vtable-vtable and in the vtables created from it. Such fields provide space for "class data". For example,

```
(define vt-of-vt (make-vtable-vtable "pw" 0))
(define vt       (make-struct vt-of-vt 0))
(struct-set! vt vtable-offset-user "my class data")
```

tail-size is the size of the tail array in the vtable-vtable itself, if *user-fields* specifies a tail array. This should be 0 if nothing extra is required or the format has no tail array. The tail array field such as '`pW`' holds the tail array size, as usual, and is followed by the extra space.

```
(define vt-vt (make-vtable-vtable "pW" 20))
(define my-vt-tail-start (1+ vtable-offset-user))
(struct-set! vt-vt (+ 3 my-vt-tail-start) "data in tail")
```

The optional *print* argument is used by `display` and `write` (etc) to print the vtable-vtable and any vtables created from it. It's called as `(print vtable port)` and should look at *vtable* and write to *port*. The default is the usual structure print function, which just gives machine addresses.

`make-struct-layout` *fields* [Scheme Procedure]
`scm_make_struct_layout` (*fields*) [C Function]

Return a structure layout symbol, from a *fields* string. *fields* is as described under `make-vtable` (see Section 6.7.9.1 [Vtables], page 214). An invalid *fields* string is an error.

```
(make-struct-layout "prpW") ⇒ prpW
(make-struct-layout "blah") ⇒ ERROR
```

`vtable-offset-user` [Scheme Variable]
`scm_vtable_offset_user` [C Macro]

The first field in a vtable which is available for application use. Such fields only exist when specified by *user-fields* in `make-vtable-vtable` above.

Here's an extended vtable-vtable example, creating classes of "balls". Each class has a "colour", which is fixed. Instances of those classes are created, and such each such ball has an "owner", which can be changed.

```
(define ball-root (make-vtable-vtable "pr" 0))

(define (make-ball-type ball-color)
  (make-struct ball-root 0
       (make-struct-layout "pw")
               (lambda (ball port)
                 (format port "#<a ~A ball owned by ~A>"
                         (color ball)
                         (owner ball)))
               ball-color))
(define (color ball)
  (struct-ref (struct-vtable ball) vtable-offset-user))
(define (owner ball)
  (struct-ref ball 0))

(define red (make-ball-type 'red))
(define green (make-ball-type 'green))

(define (make-ball type owner) (make-struct type 0 owner))

(define ball (make-ball green 'Nisse))
ball ⇒ #<a green ball owned by Nisse>
```

6.7.10 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.

6.7.11 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.

6.7.11.1 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.

6.7.11.2 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:

```
(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.

```
(set! task-list (acons 3 "pay gas bill" '()))
task-list
⇒
((3 . "pay gas bill"))

(set! task-list (acons 3 "tidy bedroom" task-list))
task-list
⇒
((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* [Scheme Procedure]
`scm_acons` (*key, value, alist*) [C Function]

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* [Scheme Procedure]
`assv-set!` *alist key value* [Scheme Procedure]
`assoc-set!` *alist key value* [Scheme Procedure]
`scm_assq_set_x` (*alist, key, val*) [C Function]
`scm_assv_set_x` (*alist, key, val*) [C Function]
`scm_assoc_set_x` (*alist, key, val*) [C Function]

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.

6.7.11.3 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* [Scheme Procedure]
`assv` *key alist* [Scheme Procedure]
`assoc` *key alist* [Scheme Procedure]
`scm_assq` (*key, alist*) [C Function]
`scm_assv` (*key, alist*) [C Function]
`scm_assoc` (*key, alist*) [C Function]

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 544).

`assq-ref` *alist key* [Scheme Procedure]
`assv-ref` *alist key* [Scheme Procedure]
`assoc-ref` *alist key* [Scheme Procedure]
`scm_assq_ref` (*alist, key*) [C Function]
`scm_assv_ref` (*alist, key*) [C Function]

`scm_assoc_ref` (*alist*, *key*) [C Function]

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.

6.7.11.4 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*) [C Function]

Delete the first entry in *alist* associated with *key*, and return the resulting alist.

6.7.11.5 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-` versions 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:

```
(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")

(sloppy-assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
#f
```

Secondly, if one of the entries in the specified alist is not a pair:

```
(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))

(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-` versions would silently cover up.

`sloppy-assq` *key alist* [Scheme Procedure]
`scm_sloppy_assq` (*key, alist*) [C Function]

Behaves like `assq` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assv` *key alist* [Scheme Procedure]
`scm_sloppy_assv` (*key, alist*) [C Function]

Behaves like `assv` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assoc` *key alist* [Scheme Procedure]
`scm_sloppy_assoc` (*key, alist*) [C Function]

Behaves like `assoc` but does not do any error checking. Recommended only for use in Guile internals.

6.7.11.6 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"))
```

6.7.12 VList-Based Hash Lists or "VHashes"

The `(ice-9 vlist)` module provides an implementation of *VList-based hash lists* (see Section 6.7.7 [VLists], page 210). 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 544).

In addition, vhashes can be manipulated using VList operations:

```
(vlist-head (vhash-consq 'a 1 vlist-null))
⇒ (a . 1)

(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 6.7.13.2 [Hash Table Reference], page 230). 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 defaults to `hash`) to compute the hash of *key*. The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the last form uses `eqv?` and `hashv`.

Note that it is important to consistently use the same hash function for *hash-proc* as was passed to `vhash-cons`. Failing to do that, the result is unpredictable.

`vhash-delete` *key vhash* [*equal?* [*hash-proc*]] [Scheme Procedure]
`vhash-delq` *key vhash* [Scheme Procedure]
`vhash-delv` *key vhash* [Scheme Procedure]

Remove all associations from *vhash* with *key*, comparing keys with *equal?* (which defaults to `equal?`), and computing the hash of *key* using *hash-proc* (which defaults to `hash`). The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the last one uses `eqv?` and `hashv`.

Again the choice of *hash-proc* must be consistent with previous calls to `vhash-cons`.

`vhash-fold` *proc init vhash* [Scheme Procedure]
`vhash-fold-right` *proc init vhash* [Scheme Procedure]

Fold over the key/value elements of *vhash* in the given direction, with each call to *proc* having the form `(`*proc* `key value result)`, where *result* is the result of the previous call to *proc* and *init* the value of *result* for the first call to *proc*.

`vhash-fold*` *proc init key vhash* [*equal?* [*hash*]] [Scheme Procedure]
`vhash-foldq*` *proc init key vhash* [Scheme Procedure]
`vhash-foldv*` *proc init key vhash* [Scheme Procedure]

Fold over all the values associated with *key* in *vhash*, with each call to *proc* having the form `(proc value result)`, where *result* is the result of the previous call to *proc* and *init* the value of *result* for the first call to *proc*.

Keys in *vhash* are hashed using *hash* are compared using *equal?*. The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the third one uses `eqv?` and `hashv`.

Example:

```
(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)
```

`alist->vhash` *alist* [*hash-proc*] [Scheme Procedure]

Return the vhash corresponding to *alist*, an association list, using *hash-proc* to compute key hashes. When omitted, *hash-proc* defaults to `hash`.

6.7.13 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 6.7.2 [Lists], page 185) for working with them.

Guile provides two types of hashtables. One is an abstract data type that can only be manipulated with the functions in this section. The other type is concrete: it uses a normal vector with alists as elements. The advantage of the abstract hash tables is that they will be automatically resized when they become too full or too empty.

6.7.13.1 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>

;; We can also use a vector of alists.
(define h (make-vector 7 '()))

h
⇒
#(() () () () () () ())

;; 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)

;; The vector now contains three elements in the alists and the frob
;; entry is at index (hashq 'frob).
h
⇒
#(((braz . "zonk")) ((foo . "bar")) () () () () ((frob . #f)))

(hashq 'frob 7)
⇒
6
```

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
```

Better is 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.

```
(hashq-get-handle h 'foo)
⇒
(foo . "bar")

(hashq-get-handle h 'not-there)
⇒
#f
```

There is no procedure for calculating the number of key/value-pairs in a hash table, but `hash-fold` can be used for doing exactly that.

```
(hash-fold (lambda (key value seed) (+ 1 seed)) 0 h)
```

```
⇒
3
```

6.7.13.2 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. The abstract kind of hash tables hide the vector in an opaque object that represents the hash table, while for the concrete kind the vector *is* the hashtable.

When the number of table entries in an abstract 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.

A abstract hash table is created with `make-hash-table`. To create a vector that is suitable as a hash table, use `(make-vector` *size* `'())`, for example.

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 6.7.11.3 [Retrieving Alist Entries], page 222). Here's an example of such functions implementing case-insensitive hashing of string keys,

```
(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 below, include `symbol-hash` (see Section 6.6.7.2 [Symbol Keys], page 170), `string-hash` and `string-hash-ci` (see Section 6.6.5.7 [String Comparison], page 147), and `char-set-hash` (see Section 6.6.4.1 [Character Set Predicates/Comparison], page 133).

`make-hash-table` [*size*] [Scheme Procedure]

Create a new abstract hash table object, with an optional minimum vector *size*.

When *size* is given, the table vector will still grow and shrink automatically, as described 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.

`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.

`hash-get-handle` *table key* [Scheme Procedure]
`hashq-get-handle` *table key* [Scheme Procedure]
`hashv-get-handle` *table key* [Scheme Procedure]
`hashx-get-handle` *hash assoc table key* [Scheme Procedure]
`scm_hash_get_handle` (*table, key*) [C Function]
`scm_hashq_get_handle` (*table, key*) [C Function]
`scm_hashv_get_handle` (*table, key*) [C Function]
`scm_hashx_get_handle` (*hash, assoc, table, key*) [C Function]

Return the `(`*`key`*` . `*`value`*`)` pair for *key* in the given hash *table*, or `#f` if *key* is not in *table*.

`hash-create-handle!` *table key init* [Scheme Procedure]
`hashq-create-handle!` *table key init* [Scheme Procedure]
`hashv-create-handle!` *table key init* [Scheme Procedure]
`hashx-create-handle!` *hash assoc table key init* [Scheme Procedure]
`scm_hash_create_handle_x` (*table, key, init*) [C Function]
`scm_hashq_create_handle_x` (*table, key, init*) [C Function]
`scm_hashv_create_handle_x` (*table, key, init*) [C Function]
`scm_hashx_create_handle_x` (*hash, assoc, table, key, init*) [C Function]

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.

`hash-map->list` *proc table* [Scheme Procedure]
`hash-for-each` *proc table* [Scheme Procedure]
`scm_hash_map_to_list` (*proc, table*) [C Function]
`scm_hash_for_each` (*proc, table*) [C Function]

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.

```
(hash-map->list cons mytable)
```

`hash-for-each-handle` *proc table* [Scheme Procedure]
`scm_hash_for_each_handle` (*proc, table*) [C Function]

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* [Scheme Procedure]
`scm_hash_fold` (*proc, init, table*) [C Function]

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)
```

6.8 Smobs

This chapter contains reference information related to defining and working with smobs. See Section 5.5 [Defining New Types (Smobs)], page 71 for a tutorial-like introduction to smobs.

`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, as described in Section 5.5 [Defining New Types (Smobs)], page 71. 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 6.18.2 [Memory Blocks], page 359).

`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 may sometimes be unnecessary because large parts of the process' memory (with the exception of `scm_gc_malloc_pointerless` regions, and `malloc`- or `scm_malloc`-allocated memory) are scanned for live pointers[4].

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`.

`void scm_set_smob_equalp` (*scm_t_bits tc, SCM* (**equalp*) (*SCM obj1, SCM obj2*)) [C Function]

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`.

---

[4] Conversely, in Guile up to the 1.8 series, the marking procedure was always required. The reason is that Guile's GC would only look for pointers in the memory area used for built-in types (the *cell heap*), not in user-allocated or statically allocated memory. This approach is often referred to as *precise marking*.

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*.

`void scm_assert_smob_type` (*scm_t_bits tag, SCM val*) [C Function]
When *val* is a smob of the type indicated by *tag*, do nothing. Else, signal an error.

`int SCM_SMOB_PREDICATE` (*scm_t_bits tag, SCM exp*) [C Macro]
Return true iff *exp* is a smob instance of the type indicated by *tag*. The expression *exp* can be evaluated more than once, so it shouldn't contain any side effects.

`void SCM_NEWSMOB` (*SCM value, scm_t_bits tag, void *data*) [C Macro]
`void SCM_NEWSMOB2` (*SCM value, scm_t_bits tag, void *data, void *data2*) [C Macro]
`void SCM_NEWSMOB3` (*SCM value, scm_t_bits tag, void *data, void *data2, void *data3*) [C Macro]
Make *value* contain a smob instance 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.

Since it is often the case (e.g., in smob constructors) that you will create a smob instance and return it, there is also a slightly specialized macro for this situation:

`SCM_RETURN_NEWSMOB` (*scm_t_bits tag, void *data*) [C Macro]
`SCM_RETURN_NEWSMOB2` (*scm_t_bits tag, void *data1, void *data2*) [C Macro]
`SCM_RETURN_NEWSMOB3` (*scm_t_bits tag, void *data1, void *data2, void *data3*) [C Macro]
This macro expands to a block of code that creates a smob instance of the type with tag *tag* and smob data *data*, *data2*, and *data3*, as with `SCM_NEWSMOB`, etc., and causes the surrounding function to return that `SCM` value. It should be the last piece of code in a block.

`scm_t_bits SCM_SMOB_FLAGS` (*SCM obj*) [C Macro]
Return the 16 extra bits of the smob *obj*. No meaning is predefined for these bits, you can use them freely.

`scm_t_bits SCM_SET_SMOB_FLAGS` (*SCM obj, scm_t_bits flags*) [C Macro]
Set the 16 extra bits of the smob *obj* to *flags*. No meaning is predefined for these bits, you can use them freely.

`scm_t_bits SCM_SMOB_DATA` (*SCM obj*) [C Macro]
`scm_t_bits SCM_SMOB_DATA_2` (*SCM obj*) [C Macro]
`scm_t_bits SCM_SMOB_DATA_3` (*SCM obj*) [C Macro]
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.

`void SCM_SET_SMOB_DATA` (*SCM obj, scm_t_bits val*) [C Macro]
`void SCM_SET_SMOB_DATA_2` (*SCM obj, scm_t_bits val*) [C Macro]

`void SCM_SET_SMOB_DATA_3` (*SCM obj, scm_t_bits val*) [C Macro]

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*) [C Macro]
`SCM SCM_SMOB_OBJECT_2` (*SCM obj*) [C Macro]
`SCM SCM_SMOB_OBJECT_3` (*SCM obj*) [C Macro]

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*) [C Macro]
`void SCM_SET_SMOB_OBJECT_2` (*SCM obj, SCM val*) [C Macro]
`void SCM_SET_SMOB_OBJECT_3` (*SCM obj, SCM val*) [C Macro]

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*) [C Macro]
`SCM * SCM_SMOB_OBJECT_2_LOC` (*SCM obj*) [C Macro]
`SCM * SCM_SMOB_OBJECT_3_LOC` (*SCM obj*) [C Macro]

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*) [Function]

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 environment 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 ...)`
: 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 ... 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 6.17.4 [Procedures for On the Fly Evaluation], page 348).

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*) [Function]

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 236 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 precomputation 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 6.17 [Read/Load/Eval/Compile], page 343, 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* [Scheme Procedure]
`scm_program_p` (*obj*) [C Function]

Returns `#t` iff *obj* is a compiled procedure.

`program-objcode` *program* [Scheme Procedure]
`scm_program_objcode` (*program*) [C Function]

Returns the object code associated with this program. See Section 10.4.6 [Bytecode and Objcode], page 769, for more information.

`program-objects` *program* [Scheme Procedure]
`scm_program_objects` (*program*) [C Function]

Returns the "object table" associated with this program, as a vector. See Section 10.3.5 [VM Programs], page 742, for more information.

`program-module` *program* [Scheme Procedure]
`scm_program_module` (*program*) [C Function]

Returns the module that was current when this program was created. Can return `#f` if the compiler could determine that this information was unnecessary.

`program-free-variables` *program* [Scheme Procedure]
`scm_program_free_variables` (*program*) [C Function]

Returns the set of free variables that this program captures in its closure, as a vector. If a closure is code with data, you can get the code from `program-objcode`, and the data via `program-free-variables`.

Some of the values captured are actually in variable "boxes". See Section 10.3.4 [Variables and the VM], page 741, for more information.

Users must not modify the returned value unless they think they're really clever.

`program-meta` *program* [Scheme Procedure]
`scm_program_meta` (*program*) [C Function]

Return the metadata thunk of *program*, or `#f` if it has no metadata.

When called, a metadata thunk returns a list of the following form: `(bindings sources arities . properties)`. The format of each of these elements is discussed below.

`program-bindings` *program* [Scheme Procedure]
`make-binding` *name boxed? index start end* [Scheme Procedure]
`binding:name` *binding* [Scheme Procedure]
`binding:boxed?` *binding* [Scheme Procedure]
`binding:index` *binding* [Scheme Procedure]
`binding:start` *binding* [Scheme Procedure]
`binding:end` *binding* [Scheme Procedure]

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 10.3.2 [VM Concepts], page 739, 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.

`program-sources` *program* [Scheme Procedure]
`source:addr` *source* [Scheme Procedure]
`source:line` *source* [Scheme Procedure]
`source:column` *source* [Scheme Procedure]
`source:file` *source* [Scheme Procedure]

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.

`program-arities` *program* [Scheme Procedure]
`scm_program_arities` (*program*) [C Function]
`program-arity` *program ip* [Scheme Procedure]
`arity:start` *arity* [Scheme Procedure]
`arity:end` *arity* [Scheme Procedure]
`arity:nreq` *arity* [Scheme Procedure]
`arity:nopt` *arity* [Scheme Procedure]
`arity:rest?` *arity* [Scheme Procedure]
`arity:kw` *arity* [Scheme Procedure]
`arity:allow-other-keys?` *arity* [Scheme Procedure]

Accessors for a representation of the "arity" of a program.

The normal case is that a procedure has one arity. For example, `(lambda (x) x)`, takes one required argument, and that's it. One could access that number of re-

quired arguments via `(arity:nreq (program-arities (lambda (x) x)))`. Similarly, `arity:nopt` gets the number of optional arguments, and `arity:rest?` returns a true value if the procedure has a rest arg.

`arity:kw` returns a list of `(kw . idx)` pairs, if the procedure has keyword arguments. The *idx* refers to the *idx*th local variable; See Section 10.3.4 [Variables and the VM], page 741, for more information. Finally `arity:allow-other-keys?` returns a true value if other keys are allowed. See Section 6.9.4 [Optional Arguments], page 240, 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.

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 syntactic 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 236).

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 procedures 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 arguments.

`lambda*` ([*var*...] [library syntax]
[#:*optional vardef*...]
[#:*key vardef*... [#:*allow-other-keys*]]
[#:*rest var* | . *var*])
body

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

```
(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 277).

`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 body2* ... [library syntax]
`let-keywords*` *args allow-other-keys?* (*binding* ...) *body1 body2* ... [library syntax]

args is evaluated and should give a list of the form `(#:keyword1 value1 #:keyword2 value2 ...)`. The *binding*s 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:

```
(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 236).

```
<case-lambda>
   --> (case-lambda <case-lambda-clause>)
<case-lambda-clause>
   --> (<formals> <definition-or-command>*)
<formals>
   --> (<identifier>*)
     | (<identifier>* . <identifier>)
     | <identifier>
```

Rest lists can be useful with `case-lambda`:

```
(define plus
  (case-lambda
    (() 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*`, but they do not contribute to the "matching" behavior. That is to say, `case-lambda*` matches only on required, optional, and rest arguments, and on the predicate; keyword arguments may be present but do not contribute to the "success" of a match. In fact a bad keyword argument list may cause an error to be raised.

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* [Scheme Procedure]

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* [Scheme Procedure]

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 ...* [Scheme Procedure]

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* [Scheme Procedure]

Return X.

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 predicates, it returns `#t` for any kind of procedure.

`procedure?` *obj* [Scheme Procedure]
`scm_procedure_p` (*obj*) [C Function]

Return `#t` if *obj* is a procedure.

`thunk?` *obj* [Scheme Procedure]
`scm_thunk_p` (*obj*) [C Function]

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.

`procedure-name` *proc* [Scheme Procedure]
`scm_procedure_name` (*proc*) [C Function]

Return the name of the procedure *proc*

`procedure-source` *proc* [Scheme Procedure]
`scm_procedure_source` (*proc*) [C Function]

Return the source of the procedure *proc*. Returns `#f` if the source code is not available.

`procedure-properties` *proc* [Scheme Procedure]
`scm_procedure_properties` (*proc*) [C Function]

Return the properties associated with *proc*, as an association list.

`procedure-property` *proc key* [Scheme Procedure]
`scm_procedure_property` (*proc, key*) [C Function]

Return the property of *proc* with name *key*.

`set-procedure-properties!` *proc alist* [Scheme Procedure]
`scm_set_procedure_properties_x` (*proc, alist*) [C Function]
Set *proc*'s property list to *alist*.

`set-procedure-property!` *proc key value* [Scheme Procedure]
`scm_set_procedure_property_x` (*proc, key, value*) [C Function]
In *proc*'s property list, set the property named *key* to *value*.

Documentation for a procedure can be accessed with the procedure `procedure-documentation`.

`procedure-documentation` *proc* [Scheme Procedure]
`scm_procedure_documentation` (*proc*) [C Function]
Return the documentation string associated with `proc`. By convention, if a procedure contains more than one expression and the first expression is a string constant, that string is assumed to contain documentation for that procedure.

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!` (REFFIXME). 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.[5]

```
(foo-ref f 0)       ⇒ bar
(foo-ref f 1)       ⇒ braz
```

Also suppose that a corresponding setter procedure called `foo-set!` does exist.

```
(foo-set! f 0 'bla)
(foo-ref f 0)       ⇒ bla
```

Now we could create a new procedure called `foo`, which is a procedure with setter, by calling `make-procedure-with-setter` with the accessor and setter procedures `foo-ref` and `foo-set!`. Let us call this new procedure `foo`.

```
(define foo (make-procedure-with-setter foo-ref foo-set!))
```

`foo` can from now an be used to either read from the data structure stored in `f`, or to write into the structure.

```
(set! (foo f 0) 'dum)
(foo f 0)           ⇒ dum
```

[5] Working definitions would be:

```
(define foo-ref vector-ref)
(define foo-set! vector-set!)
(define f (make-vector 2 #f))
```

`make-procedure-with-setter` *procedure setter* [Scheme Procedure]
`scm_make_procedure_with_setter` (*procedure, setter*) [C Function]

Create a new procedure which behaves like *procedure*, but with the associated setter *setter*.

`procedure-with-setter?` *obj* [Scheme Procedure]
`scm_procedure_with_setter_p` (*obj*) [C Function]

Return `#t` if *obj* is a procedure with an associated setter procedure.

`procedure` *proc* [Scheme Procedure]
`scm_procedure` (*proc*) [C Function]

Return the procedure of *proc*, which must be an applicable struct.

`setter` *proc* [Scheme Procedure]

Return the setter of *proc*, which must be either a procedure with setter or an operator struct.

6.9.9 Inlinable Procedures

You can define an *inlinable procedure* by using `define-inlinable` instead of `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:

```
scheme@(guile-user)> ,optimize (define (foo x)
                                 (define (bar) (+ x 3))
                                 (* (bar) 2))
$1 = (define foo
       (lambda (#{x 94}#) (* (+ #{x 94}# 3) 2)))
```

The partial evaluator does not inline top-level bindings, though, so this is a situation where you may find it interesting to use `define-inlinable`.

Procedures defined with `define-inlinable` are *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 6.25.4 [Traps], page 440). 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.

`define-inlinable` (*name parameter* . . .) *body1 body2* . . . [Scheme Syntax]

Define *name* as a procedure with parameters *parameter*s and bodies *body1*, *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 *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:

```
(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[6].

6.10.1 Defining Macros

A macro is a binding between a keyword and a syntax transformer. Since it's difficult to discuss `define-syntax` without discussing the format of transformers, consider the following example macro definition:

```
(define-syntax when
  (syntax-rules ()
    ((when condition exp ...)
     (if condition
         (begin exp ...)))))

(when #t
  (display "hey ho\n")
  (display "let's go\n"))
⊣ hey ho
⊣ let's go
```

In this example, the `when` binding is bound with `define-syntax`. Syntax transformers are discussed in more depth in Section 6.10.2 [Syntax Rules], page 249 and Section 6.10.3 [Syntax Case], page 253.

`define-syntax` *keyword transformer* [Syntax]

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* ... [Syntax]

Bind each *keyword* to its corresponding *transformer* while expanding *exp1 exp2* ....

A `let-syntax` binding only exists at expansion-time.

[6] These days such embedded languages are often referred to as *embedded domain-specific languages*, or EDSLs.

```
(let-syntax ((unless
              (syntax-rules ()
                ((unless condition exp ...)
                 (if (not condition)
                     (begin exp ...))))))
  (unless #t
    (primitive-exit 1))
  "rock rock rock")
⇒ "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* ... [Syntax]

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*.

```
(letrec-syntax ((my-or
                 (syntax-rules ()
                   ((my-or)
                    #t)
                   ((my-or exp)
                    exp)
                   ((my-or exp rest ...)
                    (let ((t exp))
                      (if exp
                          exp
                          (my-or rest ...)))))))
  (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.

`syntax-rules` *literals* (*pattern template*)... [Syntax]

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) ...) exp 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.

```
(define-syntax kwote
  (syntax-rules ()
    ((kwote exp)
     (quote exp))))
(kwote (foo . bar))
⇒ (foo . bar)
```

An improper list of patterns matches as rest arguments do:

```
(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.

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) exp ...)
     (let ((var val)) 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:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (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)
⇒ foo
```

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[7].

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")
⇒ #f
(let ((foo "foo"))
  (is-literal-foo? foo))
⇒ #f
```

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:

[7] Language lawyers probably see the need here for use of `literal-identifier=?` rather than `free-identifier=?`, and would probably be correct. Patches accepted.

```
(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:

```
(define-syntax my-or
  (syntax-rules ()
    ((my-or)
     #t)
    ((my-or exp)
     exp)
    ((my-or exp rest ...)
     (let ((t exp))
       (if exp
           exp
           (my-or rest ...))))))
```

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 260), 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 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`:

```
(define-syntax when
  (syntax-rules ()
    ((_ test e e* ...)
     (if test (begin e e* ...)))))

(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 260), `syntax-case` macros transform syntax objects, not raw Scheme forms. Recall the naive expansion of `my-or` given in the previous section:

```
(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* [Syntax]

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:

```
(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.

```
(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 `(syntax (+ 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`:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (k ...) ((keyword . pattern) template) ...)
       #'(lambda (x)
           (syntax-case x (k ...)
             ((dummy . pattern) #'template)
             ...))))))
```

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:

`identifier?` *syntax-object* [Scheme Procedure]

Returns `#t` iff *syntax-object* is an identifier.

```
;; 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:

```
;; 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`.

`datum->syntax` *for-syntax datum* [Scheme Procedure]

Create a syntax object that wraps *datum*, within the lexical context corresponding to the syntax object *for-syntax*.

For completeness, we should mention that it is possible to strip the metadata from a syntax object, returning a raw Scheme datum:

`syntax->datum` *syntax-object* [Scheme Procedure]

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:

```
;; 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:

```
;; 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 defines `quasisyntax`, and the related `unsyntax` and `unsyntax-splicing`, abbreviated by the reader as `#‘`, `#,`, and `#,@`, 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/`.

6.10.4 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` iff the syntax objects *a* and *b* refer to the same lexically-bound identifier.

`free-identifier=?` *a b* [Scheme Procedure]

Return `#t` iff the syntax objects *a* and *b* refer to the same free identifier.

`generate-temporaries` *ls* [Scheme Procedure]

Return a list of temporary identifiers as long as *ls* is long.

`syntax-source` *x* [Scheme Procedure]

Return the source properties that correspond to the syntax object *x*. See Section 6.25.2 [Source Properties], page 432, for more information.

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* [Scheme Procedure]

Resolve the identifer *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.

`pattern-variable`
A pattern variable, bound via syntax-case. The value is an opaque object, internal to the expander.

`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.

```
(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) ...))))))

(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")))
```

```
⊣ Launching missiles!
⇒ (#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 defmacros 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))
;; iff 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[8] 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 7.5.26 [SRFI-39], page 583), except that the dynamic binding only occurs during macro expansion. The code after macro expansion remains lexically scoped.

`define-syntax-parameter` *keyword transformer* [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-`

[8] Described in the paper *Keeping it Clean with Syntax Parameters* by Barzilay, Culpepper and Flatt.

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:

```
(define-syntax-parameter return
  (lambda (stx)
    (syntax-violation 'return "return used outside of a lambda^" stx)))
```

syntax-parameterize ((*keyword transformer*) ...) *exp* ... [Syntax]

Adjusts *keyword* ... to use the values obtained by evaluating their *transformer* ..., in the expansion of the *exp* ... 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* ... use the new transformers. This is somewhat similar to how parameterize adjusts the values of regular parameters, rather than creating new bindings.

```
(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
;; return immediately if it sees any 0 element.
(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 (compile 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 `eval`, `load`, and `compile`. If you need to use `eval-when`, use it with all three conditions, as in the above example. Other uses of `eval-when` may void your warranty or poison your cat.

6.10.9 Internal Macros

`make-syntax-transformer` *name type binding* [Scheme Procedure]

Construct a syntax transformer object. This is part of Guile's low-level support for syntax-case.

`macro?` *obj* [Scheme Procedure]
`scm_macro_p` (*obj*) [C Function]

Return `#t` iff *obj* is a syntax transformer.

Note that it's a bit difficult to actually get a macro as a first-class object; simply naming it (like `case`) will produce a syntax error. But it is possible to get these objects using `module-ref`:

```
(macro? (module-ref (current-module) 'case))
⇒ #t
```

`macro-type` *m* [Scheme Procedure]
`scm_macro_type` (*m*) [C Function]

Return the *type* that was given when *m* was constructed, via `make-syntax-transformer`.

`macro-name` *m* [Scheme Procedure]
`scm_macro_name` (*m*) [C Function]
Return the name of the macro *m*.

`macro-binding` *m* [Scheme Procedure]
`scm_macro_binding` (*m*) [C Function]
Return the binding of the macro *m*.

`macro-transformer` *m* [Scheme Procedure]
`scm_macro_transformer` (*m*) [C Function]
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 6.7.2.7 [List Searching], page 189).

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.

`eq?` *x y* [Scheme Procedure]
`scm_eq_p` (*x*, *y*) [C Function]
Return `#t` if *x* and *y* are the same object, except for numbers and characters. For example,

```
(define x (vector 1 2 3))
(define y (vector 1 2 3))

(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,

```
(let ((n (+ 2 3)))
  (eq? n n))       ⇒ *unspecified*
```

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?`,

```
(define x (cdr '(123)))
(define y (cdr '(456)))
(eq? x y) ⇒ #t

(define x (string->symbol "foo"))
(eq? x 'foo) ⇒ #t
```

`int scm_is_eq` (*SCM x, SCM y*) [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).

`eqv?` *x y* [Scheme Procedure]
`scm_eqv_p` (*x, y*) [C Function]

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).

```
(eqv? 3 (+ 1 2)) ⇒ #t
(eqv? 1 1.0)     ⇒ #f
```

`equal?` *x y* [Scheme Procedure]
`scm_equal_p` (*x, y*) [C Function]

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.

```
(equal? (list 1 2 3) (list 1 2 3))   ⇒ #t
(equal? (list 1 2 3) (vector 1 2 3)) ⇒ #f
```

For other objects, `equal?` compares as per `eqv?` above, which means characters and numbers are compared by type and value (and like `eqv?`, exact and inexact numbers are not `equal?`, even if their value is the same).

```
(equal? 3 (+ 1 2)) ⇒ #t
(equal? 1 1.0)     ⇒ #f
```

Hash tables are currently only compared as per `eq?`, so two different tables are not `equal?`, even if their contents are the same.

`equal?` does not support circular data structures, it may go into an infinite loop if asked to compare two circular lists or similar.

New application-defined object types (see Section 5.5 [Defining New Types (Smobs)], page 71) have an `equalp` handler which is called by `equal?`. This lets an application traverse the contents or control what is considered `equal?` for two objects of such a type. If there's no such handler, the default is to just compare as per `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 246) that can be used with the generalized form of `set!` (REFFIXME) to set and retrieve that property for any Scheme object. So, setting a property looks like this:

```
(set! (my-property obj1) value-for-obj1)
(set! (my-property obj2) value-for-obj2)
```

And retrieving values of the same property looks like this:

```
(my-property obj1)
⇒
value-for-obj1

(my-property obj2)
⇒
value-for-obj2
```

To create an object property in the first place, use the `make-object-property` procedure:

```
(define my-property (make-object-property))
```

`make-object-property` [Scheme Procedure]

Create and return an object property. An object property is a procedure-with-setter that can be called in two ways. `(set! (`*property obj*`) `*val*`)` sets *obj*'s *property* to *val*. `(`*property obj*`)` returns the current setting of *obj*'s *property*.

A single object property created by `make-object-property` can associate distinct property values with all Scheme values that are distinguishable by `eq?` (including, for example, integers).

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* [Scheme Procedure]
`scm_object_properties` (*obj*) [C Function]
Return *obj*'s property list.

`set-object-properties!` *obj alist* [Scheme Procedure]
`scm_set_object_properties_x` (*obj, alist*) [C Function]
Set *obj*'s property list to *alist*.

`object-property` *obj key* [Scheme Procedure]
`scm_object_property` (*obj, key*) [C Function]
Return the property of *obj* with name *key*.

`set-object-property!` *obj key value* [Scheme Procedure]
`scm_set_object_property_x` (*obj, key, value*) [C Function]
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* [Scheme Procedure]
`scm_merge` (*alist, blist, less*) [C Function]
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* [Scheme Procedure]
`scm_merge_x` (*alist, blist, less*) [C Function]
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.

`sorted?` *items less* [Scheme Procedure]
`scm_sorted_p` (*items, less*) [C Function]

Return `#t` iff *items* is a list or vector such that, for each element x and the next element y of *items*, `(less y x)` returns `#f`.

`sort` *items less* [Scheme Procedure]
`scm_sort` (*items, less*) [C Function]

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.

`sort!` *items less* [Scheme Procedure]
`scm_sort_x` (*items, less*) [C Function]

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.

`stable-sort` *items less* [Scheme Procedure]
`scm_stable_sort` (*items, less*) [C Function]

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.

`stable-sort!` *items less* [Scheme Procedure]
`scm_stable_sort_x` (*items, less*) [C Function]

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.

`sort-list` *items less* [Scheme Procedure]
`scm_sort_list` (*items, less*) [C Function]

Sort the list *items*, using *less* for comparing the list elements. This is a stable sort.

`sort-list!` *items less* [Scheme Procedure]
`scm_sort_list_x` (*items, less*) [C Function]

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 6.7.2 [Lists], page 185) 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 `reset-hook!`. When an application wants to run a hook, it does so using `run-hook`.

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.

```
(define hook (make-hook 2))
hook
⇒ #<hook 2 40286c90>
```

Now we are ready to add some procedures to the newly created hook with `add-hook!`. In the following example, two procedures are added, which print different messages and do different things with their arguments.

```
(add-hook! hook (lambda (x y)
                    (display "Foo: ")
                    (display (+ x y))
                    (newline)))
(add-hook! hook (lambda (x y)
                    (display "Bar: ")
                    (display (* x y))
                    (newline)))
```

Once the procedures have been added, we can invoke the hook using `run-hook`.

```
(run-hook hook 3 4)
⊣ Bar: 12
⊣ Foo: 7
```

Note that the procedures are called in the reverse of the order with which they were added. This is because the default behaviour of `add-hook!` is to add its procedure to the *front* of the hook's procedure list. You can force `add-hook!` to add its procedure to the *end* of the list instead by providing a third `#t` argument on the second call to `add-hook!`.

```
(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!

(run-hook hook 3 4)
⊣ Foo: 7
⊣ Bar: 12
```

6.11.6.2 Hook Reference

When you create a hook with `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 `make-hook`, it defaults to zero. All procedures of a given hook must have the same arity, and when the procedures are invoked using `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 `add-hook!` is omitted or is equal to `#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 `run-hook`.

The ordering of the list of procedures returned by `hook->list` matches the order in which those procedures would be called if the hook was run using `run-hook`.

Note that the C functions in the following entries are for handling *Scheme-level* hooks in C. There are also *C-level* hooks which have their own interface (see Section 6.11.6.4 [C Hooks], page 273).

`make-hook` [*n_args*] [Scheme Procedure]
`scm_make_hook` (*n_args*) [C Function]

Create a hook for storing procedure of arity *n_args*. *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 *x* is a hook, `#f` otherwise.

`hook-empty?` *hook* [Scheme Procedure]
`scm_hook_empty_p` (*hook*) [C Function]

Return `#t` if *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 *proc* to the hook *hook*. The procedure is added to the end if *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 *proc* from the hook *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 *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 *hook* to a list.

`run-hook` *hook arg* . . . [Scheme Procedure]
`scm_run_hook` (*hook, args*) [C Function]

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*) [C Function]

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*) [C Macro]

Return 1 if *x* is a Scheme-level hook, 0 otherwise.

6.11.6.3 Handling Scheme-level hooks from C code

Here is an example of how to handle Scheme-level hooks from C code using the above functions.

```
if (scm_is_true (scm_hook_p (obj)))
  /* handle Scheme-level hook using C functions */
  scm_reset_hook_x (obj);
else
  /* do something else (obj is not a hook) */
```

6.11.6.4 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` [C Type]

Data type for a C hook. The internals of this type should be treated as opaque.

`scm_t_c_hook_type` [C Enum]

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, scm_t_c_hook_type type*) [C Function]

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, scm_t_c_hook_function func, void *func_data, int appendp*) [C Function]

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, scm_t_c_hook_function func, void *func_data*) [C Function]

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.

6.11.6.5 Hooks for Garbage Collection

Whenever 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 6.21.2 [Asyncs], page 397) 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 6.18.4 [Guardians], page 364). 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.

6.11.6.6 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

```
(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* [Scheme Syntax]

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*) [C Function]
`scm_c_define` (*const char *name, value*) [C Function]

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 277) act differently (see Section 6.12.3 [Internal Definitions], page 278).

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 you there are some variables that you *don't* want to redefine all the time. For these, use `define-once`.

`define-once` *name value* [Scheme Syntax]

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 (`let`, `let*`, `letrec`, and `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 `let` expressions has a few properties which are well worth knowing.

The most basic local binding construct is `let`.

`let` *bindings body* [syntax]

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 `let` expression is evaluated as follows.

- All *init* expressions are evaluated.
- New storage is allocated for the *variables*.
- The values of the *init* expressions are stored into the variables.
- The expressions in *body* are evaluated in order, and the value of the last expression is returned as the value of the `let` expression.

The *init* expressions are not allowed to refer to any of the *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* [syntax]

Similar to `let`, but the variable bindings are performed sequentially, that means that all *init* expression are allowed to use the variables defined on their left in the binding list.

A `let*` expression can always be expressed with nested `let` expressions.

```
(let* ((a 1) (b a))
   b)
≡
(let ((a 1))
  (let ((b a))
    b))
```

`letrec` *bindings body* [syntax]

Similar to `let`, but it is possible to refer to the *variable* from lambda expression created in any of the *inits*. That is, procedures created in the *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* [syntax]

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)))
  (* a b))
⇒ ;; Error: unbound variable: a

(letrec* ((a 42)
          (b (+ a 10)))
  (* a b))
⇒ 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 283)

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 275), 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 `define`s 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*) [C Function]

Return `#t` if *sym* is defined in the module *module* or the current module when *module* is not specified; otherwise return `#f`.

6.13 Controlling the Flow of Program Execution

See Section 5.4.3 [Control Flow], page 66 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 278). 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))))
  (values seal open))
```

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 248), 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.

`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`.

`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:

```
(define-syntax-rule (when test stmt stmt* ...)
  (if test (begin stmt stmt* ...)))

(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 . . .* [syntax]

Each `cond`-clause must look like this:

```
(test expression ...)
```

where *test* and *expression* are arbitrary expression, 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 values, 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`. Iff 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 ...* [syntax]

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.

`and` *expr ...* [syntax]

Evaluate the *expr*s 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.

`or` *expr* ... [syntax]

Evaluate the *expr*s 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`.

`do` ((*variable init* [*step*]) ...) (*test expr* ...) *body* ... [syntax]

Bind *variable*s 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 *variable*s, like a new `let` for each iteration. This is done for *variable*s 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).

```
(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* . . . [syntax]

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*.

`break` *break-arg* . . . [Scheme Procedure]

Break out of the `while` form.

`continue` [Scheme Procedure]

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`.

```
(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,

```
(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 277).

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*] [Scheme Procedure]

Make a new prompt tag. Currently prompt tags are generated symbols. This may change in some future Guile version.

`default-prompt-tag` [Scheme Procedure]

Return the default prompt tag. Having a distinguished default prompt tag allows some useful prompt and abort idioms, discussed in the next section.

`abort-to-prompt` *tag val1 val2 . . .* [Scheme Procedure]

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:

```
(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:

```
(define cont
  (lambda (x)
    (+ 34 x)))
```

So, if we call `cont` with one numeric value, we get that number, incremented by 34:

```
(cont 8)
⇒ 42
(* 2 (cont 8))
⇒ 84
```

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 `(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 `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.

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:

```
(use-modules (ice-9 control))
```

Firstly, we have a helpful abbreviation for the `call-with-prompt` operator.

`%` *expr* [Scheme Syntax]
`%` *expr handler* [Scheme Syntax]
`%` *tag expr handler* [Scheme Syntax]

Evaluate *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:

```
(define (default-prompt-handler k proc)
  (% (default-prompt-tag)
     (proc k)
     default-prompt-handler))
```

The `%` symbol is chosen because it looks like a prompt.

Likewise there is an abbreviation for `abort-to-prompt`, which assumes the default prompt tag:

`abort` *val1 val2 ...* [Scheme Procedure]

Abort to the default prompt tag, passing *val1 val2 ...* 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

```
(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.

```
(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]
`call/cc` *proc* [Scheme Procedure]

Capture the current continuation and call `(proc cont)` with it. The return value is the value returned by *proc*, or when `(cont value)` is later invoked, the return is the *value* passed.

Normally *cont* should be called with one argument, but when the location resumed is expecting multiple values (see Section 6.13.7 [Multiple Values], page 290) then they should be passed as multiple arguments, for instance `(cont x y z)`.

cont may only be used from the same side of a continuation barrier as it was created (see Section 6.13.12 [Continuation Barriers], page 303), and in a multi-threaded program only from the thread in which it was created.

The call to *proc* is not part of the continuation captured, it runs only when the continuation is created. Often a program will want to store *cont* somewhere for later use; this can be done in *proc*.

The `call` in the name `call-with-current-continuation` refers to the way a call to *proc* gives the newly created continuation. It's not related to the way a call is used later to invoke that continuation.

`call/cc` is an alias for `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 `format`. The `lambda` stores this in `kont` and gives an initial return `1` which is displayed. The later invocation of `kont` resumes the captured point, but this time returning `2`, which is displayed.

When Guile is run interactively, a call to `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 `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 298) 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 285). 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* ... [Scheme Procedure]
`scm_values` (*args*) [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*) [C Function]

`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 more a efficient representation.

`size_t scm_c_nvalues` (*SCM obj*) [C Function]

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*) [C Function]

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* [Scheme Procedure]

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`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
⇒ 5

(call-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 558).

```
(use-modules (ice-9 receive))
```

`receive` *formals expr body* . . . [library syntax]

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 236). 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 535),

```
(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:

```
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.

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`.

`catch` *key thunk handler* [*pre-unwind-handler*] [Scheme Procedure]
`scm_catch_with_pre_unwind_handler` (*key, thunk, handler, pre_unwind_handler*) [C Function]
`scm_catch` (*key, thunk, handler*) [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:

```
(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:

```
(lambda (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.

`SCM scm_c_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, scm_t_catch_handler pre_unwind_handler, void *pre_unwind_handler_data*) [C Function]

`SCM scm_internal_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]

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_upto_here_1` (see Section 5.5.6 [Remembering During Operations], page 78).

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* [Scheme Procedure]

`scm_with_throw_handler` (*key, thunk, handler*) [C Function]

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 298.)

`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*) [C Function]

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 292) 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:

```
(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.

`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\"))"
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 288) 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` *msg arg . . .* [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* [Scheme Procedure]
`scm_error_scm` (*key, subr, message, args, data*) [C Function]

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` or `wrong-type-arg`, it is a list containing the bad value; otherwise it will usually be `#f`.

`strerror` *err* [Scheme Procedure]
`scm_strerror` (*err*) [C Function]

Return the Unix error message corresponding to *err*, an integer `errno` value.

When `setlocale` has been called (see Section 7.2.13 [Locales], page 501), the message is in the language and charset of `LC_MESSAGES`. (This is done by the C library.)

`false-if-exception` *expr* [syntax]

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_to_locale_string (s1);

  /* Call 'free (c_s1)' when the dynwind context is left.
  */
  scm_dynwind_unwind_handler (free, c_s1, SCM_F_WIND_EXPLICITLY);
```

```
  c_s2 = scm_to_locale_string (s2);

  /* Same as above, but more concisely.
  */
  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_take_locale_string (res);
}
```

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
`scm_dynamic_wind` (*in_guard, thunk, out_guard*) [C Function]

All three arguments must be 0-argument procedures. *in_guard* is called, then *thunk*, then *out_guard*.

If, any time during the execution of *thunk*, the dynamic extent of the `dynamic-wind` expression is escaped non-locally, *out_guard* is called. If the dynamic extent of the dynamic-wind is re-entered, *in_guard* is called. Thus *in_guard* and *out_guard* may be called any number of times.

```
(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.

`void scm_dynwind_end ()` [C Function]

End the current dynamic context explicitly and make the previous one current.

`scm_t_wind_flags` [C Type]

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.

`SCM_F_WIND_EXPLICITLY`
The registered action is also carried out when the dynwind context is entered or left locally.

`void scm_dynwind_unwind_handler` (*void* (**func*)(*void **), *void *data*, *scm_t_wind_flags flags*) [C Function]

`void scm_dynwind_unwind_handler_with_scm` (*void* (**func*)(*SCM*), *SCM data*, *scm_t_wind_flags flags*) [C Function]

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.

`void scm_dynwind_rewind_handler` (*void* (**func*)(*void **), *void *data*, *scm_t_wind_flags flags*) [C Function]

`void scm_dynwind_rewind_handler_with_scm` (*void* (**func*)(*SCM*), *SCM data*, *scm_t_wind_flags flags*) [C Function]

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.

`void scm_dynwind_free` (*void *mem*) [C Function]

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 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:

`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.

6.13.11.1 C Support

In the following C functions, *SUBR* and *MESSAGE* parameters can be `NULL` to give the effect of `#f` described above.

`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 297).

`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*) [C Function]
`void scm_out_of_range` (*char *subr, SCM bad_value*) [C Function]
`void scm_wrong_num_args` (*SCM proc*) [C Function]
`void scm_wrong_type_arg` (*char *subr, int argnum, SCM bad_value*) [C Function]
`void scm_wrong_type_arg_msg` (*char *subr, int argnum, SCM bad_value, const char *expected*) [C Function]

void scm_memory_error (*char* *subr) [C Function]

Throw an error with the various keys described above.

void scm_misc_error (*const char* *subr, *const char* *message, *SCM* args) [C Function]

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 6.14.3 [Writing], page 308), and the corresponding arguments in the *args* list.

6.13.11.2 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) [Macro]

void SCM_ASSERT_TYPE (*int* test, *SCM* obj, *unsigned int* position, *const char* *subr, *const char* *expected) [Macro]

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 [Macro]
int SCM_ARG2 [Macro]
int SCM_ARG3 [Macro]
int SCM_ARG4 [Macro]
int SCM_ARG5 [Macro]
int SCM_ARG6 [Macro]
int SCM_ARG7 [Macro]

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.

int SCM_ARGn [Macro]

Passing a value of zero or SCM_ARGn for *position* allows to leave it unspecified which argument's type is incorrect. Again, SCM_ARGn should be preferred over a raw zero constant.

6.13.12 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.

`with-continuation-barrier` *proc* [Scheme Procedure]
`scm_with_continuation_barrier` (*proc*) [C Function]

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.

`void * scm_c_with_continuation_barrier` (*void *(*func) (void *), void *data*) [C Function]

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

Sequential input/output in Scheme is represented by operations on a *port*. This chapter explains the operations that Guile provides for working with ports.

Ports are created by opening, for instance `open-file` for a file (see Section 6.14.9.1 [File Ports], page 314). Characters can be read from an input port and written to an output port, or both on an input/output port. A port can be closed (see Section 6.14.4 [Closing], page 309) when no longer required, after which any attempt to read or write is an error.

The formal definition of a port is very generic: an input port is simply "an object which can deliver characters on demand," and an output port is "an object which can accept characters." Because this definition is so loose, it is easy to write functions that simulate ports in software. *Soft ports* and *string ports* are two interesting and powerful examples of this technique. (see Section 6.14.9.3 [Soft Ports], page 318, and Section 6.14.9.2 [String Ports], page 317.)

Ports are garbage collected in the usual way (see Section 6.18 [Memory Management], page 358), and will be closed at that time if not already closed. In this case any errors occurring in the close will not be reported. Usually a program will want to explicitly close so as to be sure all its operations have been successful. Of course if a program has abandoned something due to an error or other condition then closing problems are probably not of interest.

It is strongly recommended that file ports be closed explicitly when no longer required. Most systems have 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.

Note that automatic garbage collection is triggered only by memory consumption, not by file or other resource usage, so a program cannot rely on that to keep it away from system limits. An explicit call to `gc` can of course be relied on to pick up unreferenced ports. If program flow makes it hard to be certain when to close then this may be an acceptable way to control resource usage.

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.

Each port has an associated character encoding that controls how bytes read from the port are converted to characters and string and controls how characters and strings written to the port are converted to bytes. When ports are created, they inherit their character encoding from the current locale, but, that can be modified after the port is created.

Currently, the ports only work with *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: 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.

`input-port?` *x* [Scheme Procedure]
`scm_input_port_p` (*x*) [C Function]

Return `#t` if *x* is an input port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`output-port?` *x* [Scheme Procedure]
`scm_output_port_p` (*x*) [C Function]

Return `#t` if *x* is an output port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`port?` *x* [Scheme Procedure]
`scm_port_p` (*x*) [C Function]

Return a boolean indicating whether *x* is a port. Equivalent to `(or (input-port?` *x*`) (output-port?` *x*`))`.

`set-port-encoding!` *port enc* [Scheme Procedure]
`scm_set_port_encoding_x` (*port, enc*) [C Function]

Sets the character encoding that will be used to interpret all port I/O. *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`).

`%default-port-encoding` [Scheme Variable]

A fluid containing `#f` or the name of the encoding to be used by default for newly created ports (see Section 6.21.7 [Fluids and Dynamic States], page 405). The value `#f` is equivalent to `"ISO-8859-1"`.

New ports are created with the encoding appropriate for the current locale if `setlocale` has been called or the value specified by this fluid otherwise.

`port-encoding` *port* [Scheme Procedure]
`scm_port_encoding` (*port*) [C Function]

Returns, as a string, the character encoding that *port* uses to interpret its input and output. The value `#f` is equivalent to `"ISO-8859-1"`.

`set-port-conversion-strategy!` *port sym* [Scheme Procedure]
`scm_set_port_conversion_strategy_x` (*port, sym*) [C Function]

Sets the behavior of the interpreter when outputting a character that is not representable in the port's current encoding. *sym* can be either `'error`, `'substitute`, or `'escape`. If it is `'error`, an error will be thrown when an nonconvertible character is encountered. If it is `'substitute`, then nonconvertible characters will be replaced with approximate characters, or with question marks if no approximately correct character is available. If it is `'escape`, it will appear as a hex escape when output.

If *port* is an open port, the conversion error behavior is set for that port. If it is `#f`, it is set as the default behavior for any future ports that get created in this thread.

`port-conversion-strategy` *port* [Scheme Procedure]
`scm_port_conversion_strategy` (*port*) [C Function]

Returns the behavior of the port when outputting a character that is not representable in the port's current encoding. It returns the symbol `error` if unrepresentable characters should cause exceptions, `substitute` if the port should try to replace unrepresentable characters with question marks or approximate characters, or `escape` if unrepresentable characters should be converted to string escapes.

If *port* is `#f`, then the current default behavior will be returned. New ports will have this default behavior when they are created.

`%default-port-conversion-strategy` [Scheme Variable]

The fluid that defines the conversion strategy for newly created ports, and for other conversion routines such as `scm_to_stringn`, `scm_from_stringn`, `string->pointer`, and `pointer->string`.

Its value must be one of the symbols described above, with the same semantics: `'error`, `'substitute`, or `'escape`.

When Guile starts, its value is `'substitute`.

Note that `(set-port-conversion-strategy! #f` *sym*`)` is equivalent to `(fluid-set! %default-port-conversion-strategy` *sym*`)`.

6.14.2 Reading

[Generic procedures for reading from ports.]

These procedures pertain to reading characters and strings from ports. To read general S-expressions from ports, See Section 6.17.2 [Scheme Read], page 346.

`eof-object?` *x* [Scheme Procedure]
`scm_eof_object_p` (*x*) [C Function]

Return `#t` if *x* is an end-of-file object; otherwise return `#f`.

`char-ready?` [*port*] [Scheme Procedure]
`scm_char_ready_p` (*port*) [C Function]

Return `#t` if a character is ready on input *port* and return `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on *port* is guaranteed not to hang. If *port* is a file port at end of file then `char-ready?` returns `#t`.

`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 `char-ready?` cannot be rubbed out. If `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.

`read-char` [*port*] [Scheme Procedure]
`scm_read_char` (*port*) [C Function]

Return the next character available from *port*, updating *port* to point to the following character. If no more characters are available, the end-of-file object is returned.

When *port*'s data cannot be decoded according to its character encoding, a `decoding-error` is raised and *port* points past the erroneous byte sequence.

`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 this function does not update `port-line` and `port-column` below.

`peek-char` [*port*] [Scheme Procedure]
`scm_peek_char` (*port*) [C Function]

Return the next character available from *port*, *without* updating *port* to point to the following character. If no more characters are available, the end-of-file object is returned.

The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` on the same port. The only difference is that the very next call to `read-char` or `peek-char` on that *port* will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

As for `read-char`, a `decoding-error` may be raised if such a situation occurs. However, unlike with `read-char`, *port* still points at the beginning of the erroneous byte sequence when the error is raised.

`unread-char` *cobj* [*port*] [Scheme Procedure]
`scm_unread_char` (*cobj, port*) [C Function]

Place character *cobj* 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. If *port* is not supplied, the current input port is used.

`unread-string` *str port* [Scheme Procedure]
`scm_unread_string` (*str, port*) [C Function]

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. If *port* is not supplied, the `current-input-port` is used.

`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.
```

Draining the buffers may be useful for cleanly finishing buffered I/O so that the file descriptor can be used directly for further input.

`port-column` *port* [Scheme Procedure]
`port-line` *port* [Scheme Procedure]
`scm_port_column` (*port*) [C Function]
`scm_port_line` (*port*) [C Function]

Return the current column number or line number of *port*. If the number is unknown, the result is #f. Otherwise, the result is a 0-origin integer - i.e. the first character of the first line is line 0, column 0. (However, when you display a file position, for example in an error message, we recommend you add 1 to get 1-origin integers. This is because lines and column numbers traditionally start with 1, and that is what non-programmers will find most natural.)

`set-port-column!` *port column* [Scheme Procedure]
`set-port-line!` *port line* [Scheme Procedure]
`scm_set_port_column_x` (*port, column*) [C Function]
`scm_set_port_line_x` (*port, line*) [C Function]

Set the current column or line number of *port*.

6.14.3 Writing

[Generic procedures for writing to ports.]

These procedures are for writing characters and strings to ports. For more information on writing arbitrary Scheme objects to ports, See Section 6.17.3 [Scheme Write], page 347.

`get-print-state` *port* [Scheme Procedure]
`scm_get_print_state` (*port*) [C Function]

Return the print state of the port *port*. If *port* has no associated print state, `#f` is returned.

`newline` [*port*] [Scheme Procedure]
`scm_newline` (*port*) [C Function]

Send a newline to *port*. If *port* is omitted, send to the current output port.

`port-with-print-state` *port* [*pstate*] [Scheme Procedure]
`scm_port_with_print_state` (*port, pstate*) [C Function]

Create a new port which behaves like *port*, but with an included print state *pstate*. *pstate* is optional. If *pstate* isn't supplied and *port* already has a print state, the old print state is reused.

`simple-format` *destination message . args* [Scheme Procedure]
`scm_simple_format` (*destination, message, args*) [C Function]

Write *message* to *destination*, defaulting to the current output port. *message* can contain `~A` (was `%s`) and `~S` (was `%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.

`write-char` *chr* [*port*] [Scheme Procedure]
`scm_write_char` (*chr, port*) [C Function]

Send character *chr* to *port*.

`void scm_c_write` (*SCM port, const void *buffer, size_t size*) [C Function]

Write *size* bytes at *buffer* to *port*.

Note that this function does not update `port-line` and `port-column` (see Section 6.14.2 [Reading], page 306).

`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 are passed to the underlying port implementation (e.g., in the case of fports, the data will be written to the file and the output buffer will be cleared.) It has no effect on an unbuffered port.

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.

6.14.4 Closing

`close-port` *port* [Scheme Procedure]
`scm_close_port` (*port*) [C Function]

Close the specified port object. Return `#t` if it successfully closes a port or `#f` if it was already closed. An exception may be raised if an error occurs, for example when flushing buffered output. See also Section 7.2.2 [Ports and File Descriptors], page 453, for a procedure which can close file descriptors.

`close-input-port` *port* [Scheme Procedure]
`close-output-port` *port* [Scheme Procedure]
`scm_close_input_port` (*port*) [C Function]
`scm_close_output_port` (*port*) [C Function]

Close the specified input or output *port*. An exception may be raised if an error occurs while closing. If *port* is already closed, nothing is done. The return value is unspecified.

See also Section 7.2.2 [Ports and File Descriptors], page 453, for a procedure which can close file descriptors.

`port-closed?` *port* [Scheme Procedure]
`scm_port_closed_p` (*port*) [C Function]

Return `#t` if *port* is closed or `#f` if it is open.

6.14.5 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*, which is interpreted according to the value of *whence*.

One of the following variables should be supplied for *whence*:

`SEEK_SET` [Variable]

Seek from the beginning of the file.

`SEEK_CUR` [Variable]

Seek from the current position.

`SEEK_END` [Variable]

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 the file. This means that the current position of a port can be obtained using:

```
(seek port 0 SEEK_CUR)
```

`ftell` *fd_port* [Scheme Procedure]
`scm_ftell` (*fd_port*) [C Function]

Return an integer representing the current position of *fd_port*, measured from the beginning. Equivalent to:

```
(seek port 0 SEEK_CUR)
```

`truncate-file` *file* [*length*] [Scheme Procedure]
`scm_truncate_file` (*file, length*) [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.

6.14.6 Line Oriented and Delimited Text

The delimited-I/O module can be accessed with:

```
(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. It's similar to the `(scsh rdelim)` module from guile-scsh, but does not use multiple values or character sets and has an extra procedure `write-line`.

`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.

Like `read-char`, this procedure can throw to `decoding-error` (see Section 6.14.2 [Reading], page 306).

`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*] [*end*] [Scheme Procedure]

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.

`write-line` *obj* [*port*] [Scheme Procedure]
`scm_write_line` (*obj, port*) [C Function]

Display *obj* and a newline character to *port*. If *port* is not specified, `(current-output-port)` is used. This function is equivalent to:

```
(display obj [port])
(newline [port])
```

Some of the aforementioned I/O functions rely on the following C primitives. These will mainly be of interest to people hacking Guile internals.

`%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>)`.

6.14.7 Block reading and writing

The Block-string-I/O module can be accessed with:

```
(use-modules (ice-9 rw))
```

It currently contains procedures that help to implement the `(scsh rw)` module in guile-scsh.

`read-string!/partial` *str* [*port_or_fdes* [*start* [*end*]]] [Scheme Procedure]
`scm_read_string_x_partial` (*str, port_or_fdes, start, end*) [C Function]

Read characters from a port or file descriptor into a string *str*. A port must have an underlying file descriptor — a so-called fport. This procedure is scsh-compatible and can efficiently read large strings. It will:

- attempt to fill the entire string, unless the *start* and/or *end* arguments are supplied. i.e., *start* defaults to 0 and *end* defaults to `(string-length str)`
- use the current input port if *port_or_fdes* is not supplied.
- return fewer than the requested number of characters in some cases, e.g., on end of file, if interrupted by a signal, or if not all the characters are immediately available.
- wait indefinitely for some input if no characters are currently available, unless the port is in non-blocking mode.
- read characters from the port's input buffers if available, instead from the underlying file descriptor.
- return `#f` if end-of-file is encountered before reading any characters, otherwise return the number of characters read.
- return 0 if the port is in non-blocking mode and no characters are immediately available.
- return 0 if the request is for 0 bytes, with no end-of-file check.

`write-string/partial` *str* [*port_or_fdes* [*start* [*end*]]] [Scheme Procedure]
`scm_write_string_partial` (*str*, *port_or_fdes*, *start*, *end*) [C Function]

Write characters from a string *str* to a port or file descriptor. A port must have an underlying file descriptor — a so-called fport. This procedure is scsh-compatible and can efficiently write large strings. It will:

- attempt to write the entire string, unless the *start* and/or *end* arguments are supplied. i.e., *start* defaults to 0 and *end* defaults to `(string-length str)`
- use the current output port if *port_of_fdes* is not supplied.
- in the case of a buffered port, store the characters in the port's output buffer, if all will fit. If they will not fit then any existing buffered characters will be flushed before attempting to write the new characters directly to the underlying file descriptor. If the port is in non-blocking mode and buffered characters can not be flushed immediately, then an `EAGAIN` system-error exception will be raised (Note: scsh does not support the use of non-blocking buffered ports.)
- write fewer than the requested number of characters in some cases, e.g., if interrupted by a signal or if not all of the output can be accepted immediately.
- wait indefinitely for at least one character from *str* to be accepted by the port, unless the port is in non-blocking mode.
- return the number of characters accepted by the port.
- return 0 if the port is in non-blocking mode and can not accept at least one character from *str* immediately
- return 0 immediately if the request size is 0 bytes.

6.14.8 Default Ports for Input, Output and Errors

`current-input-port` [Scheme Procedure]
`scm_current_input_port` () [C Function]

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` [Scheme Procedure]
`scm_current_output_port` () [C Function]

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 6.14.3 [Writing], page 308) at selected points.

`current-error-port` [Scheme Procedure]
`scm_current_error_port ()` [C Function]

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` *port* [Scheme Procedure]
`set-current-output-port` *port* [Scheme Procedure]
`set-current-error-port` *port* [Scheme Procedure]
`scm_set_current_input_port` (*port*) [C Function]
`scm_set_current_output_port` (*port*) [C Function]
`scm_set_current_error_port` (*port*) [C Function]

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.

`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 298). 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.

6.14.9 Types of Port

[Types of port; how to make them.]

6.14.9.1 File Ports

The following procedures are used to open file ports. See also Section 7.2.2 [Ports and File Descriptors], page 453, for an interface to the Unix `open` system call.

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 304).

`open-file` *filename mode* [Scheme Procedure]
`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 453

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 7.2.2 [Ports and File Descriptors], page 453

'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 any coding declaration or port encoding. See Section 6.14.1 [Ports], page 304, 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 6.14.10.8 [R6RS Binary Input], page 326, and Section 6.14.10.11 [R6RS Binary Output], page 331, 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.

If a file cannot be opened with the access requested, open-file throws an exception.

When the file is opened, this procedure will scan for a coding declaration (see Section 6.17.8 [Character Encoding of Source Files], page 355). If a coding declaration is found, it will be used to interpret the file. Otherwise, the port's encoding will be used. To suppress this behavior, open the file in binary mode and then set the port encoding explicitly using set-port-encoding!.

In theory we could create read/write ports which were buffered in one direction only. However this isn't included in the current interfaces.

open-input-file *filename* [Scheme Procedure]

Open *filename* for input. Equivalent to

```
(open-file filename "r")
```

open-output-file *filename* [Scheme Procedure]

Open *filename* for output. Equivalent to

```
(open-file filename "w")
```

`call-with-input-file` *filename proc* [Scheme Procedure]
`call-with-output-file` *filename proc* [Scheme Procedure]

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]
`with-output-to-file` *filename thunk* [Scheme Procedure]
`with-error-to-file` *filename thunk* [Scheme Procedure]

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 (eg. 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* [Scheme Procedure]
`scm_file_port_p` (*obj*) [C Function]

Determine whether *obj* is a port that is related to a file.

6.14.9.2 String Ports

The following allow string ports to be opened by analogy to R4RS file port facilities:

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.

`call-with-output-string` *proc* [Scheme Procedure]
`scm_call_with_output_string` (*proc*) [C Function]

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.

Note that which characters can be written to a string port depend on the port's encoding. The default encoding of string ports is specified by the `%default-port-encoding` fluid (see Section 6.14.1 [Ports], page 304). For instance, it is an error to write Greek letter alpha to an ISO-8859-1-encoded string port since this character cannot be represented with ISO-8859-1:

```
(define alpha (integer->char #x03b1)) ; GREEK SMALL LETTER ALPHA

(with-fluids ((%default-port-encoding "ISO-8859-1"))
  (call-with-output-string
    (lambda (p)
      (display alpha p))))

⇒
Throw to key ‘encoding-error’
```

Changing the string port's encoding to a Unicode-capable encoding such as UTF-8 solves the problem.

`call-with-input-string` *string proc* [Scheme Procedure]
`scm_call_with_input_string` (*string, proc*) [C Function]

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* [Scheme Procedure]

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.

See `call-with-output-string` above for character encoding considerations.

`with-input-from-string` *string thunk* [Scheme Procedure]

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` *str* [Scheme Procedure]
`scm_open_input_string` (*str*) [C Function]

Take a string and return an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`open-output-string` [Scheme Procedure]
`scm_open_output_string` () [C Function]

Return an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`get-output-string` *port* [Scheme Procedure]
`scm_get_output_string` (*port*) [C Function]

Given an output port created by `open-output-string`, return a string consisting of the characters that have been output to the port so far.

`get-output-string` must be used before closing *port*, once closed the string cannot be obtained.

A string port can be used in many procedures which accept a port but which are not dependent on implementation details of fports. E.g., seeking and truncating will work on a string port, but trying to extract the file descriptor number will fail.

6.14.9.3 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]
`scm_make_soft_port` (*pv, modes*) [C Function]

Return a port capable of receiving or delivering characters as specified by the *modes* string (see Section 6.14.9.1 [File Ports], page 314). *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:

```
(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>
```

6.14.9.4 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]
`scm_sys_make_void_port` (*mode*) [C Function]

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 6.14.9.1 [File Ports], page 314.

6.14.10 R6RS I/O Ports

The I/O port API of the Revised Report^6 on the Algorithmic Language Scheme (R6RS) (`http://www.r6rs.org/`) is provided by the `(rnrs io ports)` module. It provides features, such as binary I/O and Unicode string I/O, that complement or refine Guile's historical port API presented above (see Section 6.14 [Input and Output], page 304). 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.

The text in this section is taken from the R6RS standard libraries document, with only minor adaptions for inclusion in this manual. The Guile developers offer their thanks to the R6RS editors for having provided the report's text under permissive conditions making this possible.

Note: The implementation of this R6RS API is not complete yet.

A subset of the `(rnrs io ports)` module is provided by the `(ice-9 binary-ports)` module. It contains binary input/output procedures and does not rely on R6RS support.

6.14.10.1 File Names

Some of the procedures described in this chapter accept a file name as an argument. Valid values for such a file name include strings that name a file using the native notation of file system paths on an implementation's underlying operating system, and may include implementation-dependent values as well.

A *filename* parameter name means that the corresponding argument must be a file name.

6.14.10.2 File Options

When opening a file, the various procedures in this library 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 7.6.2.24 [rnrs enums], page 623) over the symbols constituting valid file options.

A *file-options* parameter name means that the corresponding argument must be a file-options object.

`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-symbol*s; they have implementation-specific meaning, if any.

Note: Only the name of *file-options-symbol* is significant.

6.14.10.3 Buffer Modes

Each port has an associated buffer mode. For an output port, the buffer mode defines when an output operation flushes the buffer associated with the output port. For an input port, the buffer mode defines how much data will be read to satisfy read operations. The possible buffer modes are the symbols `none` for no buffering, `line` for flushing upon line endings and reading up to line endings, or other implementation-dependent behavior, and `block` for arbitrary buffering. This section uses the parameter name *buffer-mode* for arguments that must be buffer-mode symbols.

If two ports are connected to the same mutable source, both ports are unbuffered, and reading a byte or character from that shared source via one of the two ports would change

the bytes or characters seen via the other port, a lookahead operation on one port will render the peeked byte or character inaccessible via the other port, while a subsequent read operation on the peeked port will see the peeked byte or character even though the port is otherwise unbuffered.

In other words, the semantics of buffering is defined in terms of side effects on shared mutable sources, and a lookahead operation has the same side effect on the shared source as a read operation.

`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.

Note: Only the name of *buffer-mode-symbol* is significant.

`buffer-mode?` *obj* [Scheme Procedure]

Returns `#t` if the argument is a valid buffer-mode symbol, and returns `#f` otherwise.

6.14.10.4 Transcoders

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 `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.

`latin-1-codec` [Scheme Procedure]
`utf-8-codec` [Scheme Procedure]
`utf-16-codec` [Scheme Procedure]

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 `eqv?` to the result of any other call to the same procedure.

`eol-style` *`eol-style-symbol`* [Scheme Syntax]

eol-style-symbol should be a symbol whose name is one of `lf`, `cr`, `crlf`, `nel`, `crnel`, `ls`, and `none`.

The form evaluates to the corresponding symbol. If the name of *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 *eol-style* argument to `make-transcoder`. Otherwise, an exception is raised.

All eol-style symbols except `none` describe a specific line-ending encoding:

`lf`	linefeed
`cr`	carriage return
`crlf`	carriage return, linefeed
`nel`	next line
`crnel`	carriage return, next line
`ls`	line separator

For a textual port with a transcoder, and whose transcoder has an eol-style symbol `none`, no conversion occurs. For a textual input port, any eol-style symbol other than `none` means that all of the above line-ending encodings are recognized and are translated into a single linefeed. For a textual output port, `none` and `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` [Scheme Procedure]

Returns the default end-of-line style of the underlying platform, e.g., `lf` on Unix and `crlf` on Windows.

`&i/o-decoding` [Condition Type]
`make-i/o-decoding-error` *port* [Scheme Procedure]
`i/o-decoding-error?` *obj* [Scheme Procedure]

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` [Condition Type]
`make-i/o-encoding-error` *port char* [Scheme Procedure]
`i/o-encoding-error?` *obj* [Scheme Procedure]
`i/o-encoding-error-char` *condition* [Scheme Procedure]

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`* [Scheme Syntax]

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.

`make-transcoder` *codec* [Scheme Procedure]
`make-transcoder` *codec eol-style* [Scheme Procedure]
`make-transcoder` *codec eol-style handling-mode* [Scheme Procedure]

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.

`native-transcoder` [Scheme procedure]

Returns an implementation-dependent transcoder that represents a possibly locale-dependent "native" transcoding.

`transcoder-codec` *transcoder* [Scheme Procedure]
`transcoder-eol-style` *transcoder* [Scheme Procedure]
`transcoder-error-handling-mode` *transcoder* [Scheme Procedure]

These are accessors for transcoder objects; when applied to a transcoder returned by `make-transcoder`, they return the *codec*, *eol-style*, and *handling-mode* arguments, respectively.

`bytevector->string` *bytevector transcoder* [Scheme Procedure]
Returns the string that results from transcoding the *bytevector* according to the input direction of the transcoder.

`string->bytevector` *string transcoder* [Scheme Procedure]
Returns the bytevector that results from transcoding the *string* according to the output direction of the transcoder.

6.14.10.5 The End-of-File Object

R5RS' `eof-object?` procedure is provided by the `(rnrs io ports)` module:

`eof-object?` *obj* [Scheme Procedure]
`scm_eof_object_p` (*obj*) [C Function]
Return true if *obj* is the end-of-file (EOF) object.

In addition, the following procedure is provided:

`eof-object` [Scheme Procedure]
`scm_eof_object` () [C Function]
Return the end-of-file (EOF) object.

```
(eof-object? (eof-object))
⇒ #t
```

6.14.10.6 Port Manipulation

The procedures listed below operate on any kind of R6RS I/O port.

`port?` *obj* [Scheme Procedure]
Returns `#t` if the argument is a port, and returns `#f` otherwise.

`port-transcoder` *port* [Scheme Procedure]
Returns the transcoder associated with *port* if *port* is textual and has an associated transcoder, and returns `#f` if *port* is binary or does not have an associated transcoder.

`binary-port?` *port* [Scheme Procedure]
Return `#t` if *port* is a *binary port*, suitable for binary data input/output.

Note that internally Guile does not differentiate between binary and textual ports, unlike the R6RS. Thus, this procedure returns true when *port* does not have an associated encoding—i.e., when `(port-encoding port)` is `#f` (see Section 6.14.1 [Ports], page 304). This is the case for ports returned by R6RS procedures such as `open-bytevector-input-port` and `make-custom-binary-output-port`.

However, Guile currently does not prevent use of textual I/O procedures such as `display` or `read-char` with binary ports. Doing so "upgrades" the port from binary to textual, under the ISO-8859-1 encoding. Likewise, Guile does not prevent use of `set-port-encoding!` on a binary port, which also turns it into a "textual" port.

`textual-port?` *port* [Scheme Procedure]
Always return `#t`, as all ports can be used for textual I/O in Guile.

`transcoded-port` *binary-port transcoder* [Scheme Procedure]

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* [Scheme Procedure]

If *port* supports it (see below), return the offset (an integer) indicating where the next octet will be read from/written to in *port*. If *port* does not support this operation, an error condition is raised.

This is similar to Guile's `seek` procedure with the `SEEK_CUR` argument (see Section 6.14.5 [Random Access], page 310).

`port-has-port-position?` *port* [Scheme Procedure]

Return `#t` is *port* supports `port-position`.

`set-port-position!` *port offset* [Scheme Procedure]

If *port* supports it (see below), set the position where the next octet will be read from/written to *port* to *offset* (an integer). If *port* does not support this operation, an error condition is raised.

This is similar to Guile's `seek` procedure with the `SEEK_SET` argument (see Section 6.14.5 [Random Access], page 310).

`port-has-set-port-position!?` *port* [Scheme Procedure]

Return `#t` is *port* supports `set-port-position!`.

`call-with-port` *port proc* [Scheme Procedure]

Call *proc*, passing it *port* and closing *port* upon exit of *proc*. Return the return values of *proc*.

6.14.10.7 Input Ports

`input-port?` *obj* [Scheme Procedure]

Returns `#t` if the argument is an input port (or a combined input and output port), and returns `#f` otherwise.

`port-eof?` *input-port* [Scheme Procedure]

Returns `#t` if the `lookahead-u8` procedure (if *input-port* is a binary port) or the `lookahead-char` procedure (if *input-port* is a textual port) would return the end-of-file object, and `#f` otherwise. The operation may block indefinitely if no data is available but the port cannot be determined to be at end of file.

`open-file-input-port` *filename* [Scheme Procedure]
`open-file-input-port` *filename file-options* [Scheme Procedure]
`open-file-input-port` *filename file-options buffer-mode* [Scheme Procedure]
`open-file-input-port` *filename file-options buffer-mode maybe-transcoder* [Scheme Procedure]

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 (see Section 6.14.10.2 [R6RS File Options], page 320), 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).

`standard-input-port` [Scheme Procedure]

Returns a fresh binary input port connected to standard input. Whether the port supports the `port-position` and `set-port-position!` operations is implementation-dependent.

`current-input-port` [Scheme Procedure]

This returns a default textual port for input. Normally, this default port is associated with standard input, but can be dynamically re-assigned using the `with-input-from-file` procedure from the `io simple (6)` library (see Section 7.6.2.16 [rnrs io simple], page 612). The port may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent.

6.14.10.8 Binary Input

R6RS binary input ports can be created with the procedures described below.

`open-bytevector-input-port` *bv* [*transcoder*] [Scheme Procedure]
`scm_open_bytevector_input_port` (*bv, transcoder*) [C Function]

Return an input port whose contents are drawn from bytevector *bv* (see Section 6.6.6 [Bytevectors], page 162).

The *transcoder* argument is currently not supported.

`make-custom-binary-input-port` *id read! get-position set-position! close* [Scheme Procedure]
`scm_make_custom_binary_input_port` (*id, read!, get-position, set-position!, close*) [C Function]

Return a new custom binary input port[9] named *id* (a string) whose input is drained by invoking *read!* and passing it a bytevector, an index where bytes should be written, and the number of bytes to read. The `read!` procedure must return an integer indicating the number of bytes read, or `0` to indicate the end-of-file.

Optionally, if *get-position* is not `#f`, it must be a thunk that will be called when `port-position` is invoked on the custom binary port and should return an integer indicating the position within the underlying data stream; if *get-position* was not supplied, the returned port does not support `port-position`.

Likewise, if *set-position!* is not `#f`, it should be a one-argument procedure. When `set-port-position!` is invoked on the custom binary input port, *set-position!* is passed an integer indicating the position of the next byte is to read.

Finally, if *close* is not `#f`, it must be a thunk. It is invoked when the custom binary input port is closed.

Using a custom binary input port, the `open-bytevector-input-port` procedure could be implemented as follows:

```
(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!))

(read (open-bytevector-input-port (string->utf8 "hello")))
⇒ hello
```

Binary input is achieved using the procedures below:

[9] This is similar in spirit to Guile's *soft ports* (see Section 6.14.9.3 [Soft Ports], page 318).

`get-u8` *port* [Scheme Procedure]
`scm_get_u8` (*port*) [C Function]

Return an octet read from *port*, a binary input port, blocking as necessary, or the end-of-file object.

`lookahead-u8` *port* [Scheme Procedure]
`scm_lookahead_u8` (*port*) [C Function]

Like `get-u8` but does not update *port*'s position to point past the octet.

`get-bytevector-n` *port count* [Scheme Procedure]
`scm_get_bytevector_n` (*port, count*) [C Function]

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]
`scm_get_bytevector_n_x` (*port, bv, start, count*) [C Function]

Read *count* bytes from *port* and store them in *bv* starting at index *start*. Return either the number of bytes actually read or the end-of-file object.

`get-bytevector-some` *port* [Scheme Procedure]
`scm_get_bytevector_some` (*port*) [C Function]

Read from *port*, blocking as necessary, until data are available or and end-of-file is reached. Return either a new bytevector containing the data read or the end-of-file object.

`get-bytevector-all` *port* [Scheme Procedure]
`scm_get_bytevector_all` (*port*) [C Function]

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).

6.14.10.9 Textual Input

`get-char` *textual-input-port* [Scheme Procedure]

Reads from *textual-input-port*, blocking as necessary, until a complete character is available from *textual-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.

`lookahead-char` *textual-input-port* [Scheme Procedure]

The `lookahead-char` procedure is like `get-char`, but it does not update *textual-input-port* to point past the character.

`get-string-n` *textual-input-port count* [Scheme Procedure]

count must be an exact, non-negative integer object, representing the number of characters to be read.

The `get-string-n` procedure reads from *textual-input-port*, blocking as necessary, until *count* characters are available, or until an end of file is reached.

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!` *textual-input-port string start count* [Scheme Procedure]

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.

The `get-string-n!` procedure reads from *textual-input-port* in the same manner as `get-string-n`. 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` *textual-input-port count* [Scheme Procedure]

Reads from *textual-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` *textual-input-port* [Scheme Procedure]

Reads from *textual-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.

> **Note:** The end-of-line style, if not `none`, will cause all line endings to be read as linefeed characters. See Section 6.14.10.4 [R6RS Transcoders], page 321.

`get-datum` *textual-input-port count* [Scheme Procedure]

Reads an external representation from *textual-input-port* and returns the datum it represents. The `get-datum` procedure returns the next datum that can be parsed from the given *textual-input-port*, updating *textual-input-port* to point exactly past the end of the external representation of the object.

Any *interlexeme space* (comment or whitespace, see Section 6.17.1 [Scheme Syntax], page 343) in the input is first skipped. If an end of file occurs after the interlexeme

space, the end-of-file object (see Section 6.14.10.5 [R6RS End-of-File], page 324) is returned.

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.

6.14.10.10 Output Ports

`output-port?` *obj* [Scheme Procedure]

Returns `#t` if the argument is an output port (or a combined input and output port), `#f` otherwise.

`flush-output-port` *port* [Scheme Procedure]

Flushes any buffered output from the buffer of *output-port* to the underlying file, device, or object. The `flush-output-port` procedure returns an unspecified values.

`open-file-output-port` *filename* [Scheme Procedure]
`open-file-output-port` *filename file-options* [Scheme Procedure]
`open-file-output-port` *filename file-options buffer-mode* [Scheme Procedure]
`open-file-output-port` *filename file-options buffer-mode maybe-transcoder* [Scheme Procedure]

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 (see Section 6.14.10.2 [R6RS File Options], page 320), 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).

`standard-output-port` [Scheme Procedure]
`standard-error-port` [Scheme Procedure]

Returns a fresh 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-output-port` [Scheme Procedure]
`current-error-port` [Scheme Procedure]

These return default textual ports for regular output and error output. Normally, these default ports are associated with standard output, and standard error, respectively. The return value of `current-output-port` can be dynamically re-assigned

using the `with-output-to-file` procedure from the `io simple (6)` library (see Section 7.6.2.16 [rnrs io simple], page 612). A port returned by one of these procedures may or may not have an associated transcoder; if it does, the transcoder is implementation-dependent.

6.14.10.11 Binary Output

Binary output ports can be created with the procedures below.

`open-bytevector-output-port` [*transcoder*] [Scheme Procedure]
`scm_open_bytevector_output_port` (*transcoder*) [C Function]

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.

```
(call-with-values
  (lambda ()
    (open-bytevector-output-port))
  (lambda (port get-bytevector)
    (display "hello" port)
    (get-bytevector)))

⇒ #vu8(104 101 108 108 111)
```

The *transcoder* argument is currently not supported.

`make-custom-binary-output-port` *id write! get-position set-position! close* [Scheme Procedure]
`scm_make_custom_binary_output_port` (*id, write!, get-position, set-position!, close*) [C Function]

Return a new custom binary output port named *id* (a string) whose output is sunk by invoking *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 `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 `make-custom-binary-input-port` (see Section 6.14.10.8 [R6RS Binary Input], page 326).

Writing to a binary output port can be done using the following procedures:

`put-u8` *port octet* [Scheme Procedure]
`scm_put_u8` (*port, octet*) [C Function]

Write *octet*, an integer in the 0–255 range, to *port*, a binary output port.

`put-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
`scm_put_bytevector` (*port, bv, start, count*) [C Function]

Write the contents of *bv* to *port*, optionally starting at index *start* and limiting to *count* octets.

6.14.10.12 Textual Output

`put-char` *port char* [Scheme Procedure]

Writes *char* to the port. The `put-char` procedure returns

`put-string` *port string* [Scheme Procedure]
`put-string` *port string start* [Scheme Procedure]
`put-string` *port string start count* [Scheme Procedure]

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*$. The `put-string` procedure writes the *count* characters of *string* starting at index *start* to the port. The `put-string` procedure returns an unspecified value.

`put-datum` *textual-output-port datum* [Scheme Procedure]

datum should be a datum value. The `put-datum` procedure writes an external representation of *datum* to *textual-output-port*. The specific external representation is implementation-dependent. However, whenever possible, an implementation should produce a representation for which `get-datum`, when reading the representation, will return an object equal (in the sense of `equal?`) to *datum*.

> **Note:** Not all datums may allow producing an external representation for which `get-datum` will produce an object that is equal to the original. Specifically, NaNs contained in *datum* may make this impossible.
>
> **Note:** The `put-datum` procedure merely writes the external representation, but no trailing delimiter. If `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 `get-datum`.

6.14.11 Using and Extending Ports in C

6.14.11.1 C Port Interface

This section describes how to use Scheme ports from C.

Port basics

There are two main data structures. A port type object (ptob) is of type `scm_ptob_descriptor`. A port instance is of type `scm_port`. Given an `SCM` variable which points to a port, the corresponding C port object can be obtained using the `SCM_PTAB_ENTRY` macro. The ptob can be obtained by using `SCM_PTOBNUM` to give an index into the `scm_ptobs` global array.

Port buffers

An input port always has a read buffer and an output port always has a write buffer. However the size of these buffers is not guaranteed to be more than one byte (e.g., the `shortbuf` field in `scm_port` which is used when no other buffer is allocated). The way in which the buffers are allocated depends on the implementation of the ptob. For example in the case of an fport, buffers may be allocated with malloc when the port is created, but in the case of an strport the underlying string is used as the buffer.

The `rw_random` flag

Special treatment is required for ports which can be seeked at random. Before various operations, such as seeking the port or changing from input to output on a bidirectional port or vice versa, the port implementation must be given a chance to update its state. The write buffer is updated by calling the `flush` ptob procedure and the input buffer is updated by calling the `end_input` ptob procedure. In the case of an fport, `flush` causes buffered output to be written to the file descriptor, while `end_input` causes the descriptor position to be adjusted to account for buffered input which was never read.

The special treatment must be performed if the `rw_random` flag in the port is non-zero.

The `rw_active` variable

The `rw_active` variable in the port is only used if `rw_random` is set. It's defined as an enum with the following values:

`SCM_PORT_READ`
: the read buffer may have unread data.

`SCM_PORT_WRITE`
: the write buffer may have unwritten data.

`SCM_PORT_NEITHER`
: neither the write nor the read buffer has data.

Reading from a port.

To read from a port, it's possible to either call existing libguile procedures such as `scm_getc` and `scm_read_line` or to read data from the read buffer directly. Reading from the buffer involves the following steps:

1. Flush output on the port, if `rw_active` is `SCM_PORT_WRITE`.
2. Fill the read buffer, if it's empty, using `scm_fill_input`.
3. Read the data from the buffer and update the read position in the buffer. Steps 2) and 3) may be repeated as many times as required.
4. Set rw_active to `SCM_PORT_READ` if `rw_random` is set.
5. update the port's line and column counts.

Writing to a port.

To write data to a port, calling `scm_lfwrite` should be sufficient for most purposes. This takes care of the following steps:

1. End input on the port, if `rw_active` is `SCM_PORT_READ`.
2. Pass the data to the ptob implementation using the `write` ptob procedure. The advantage of using the ptob `write` instead of manipulating the write buffer directly is that it allows the data to be written in one operation even if the port is using the single-byte `shortbuf`.
3. Set `rw_active` to `SCM_PORT_WRITE` if `rw_random` is set.

6.14.11.2 Port Implementation

This section describes how to implement a new port type in C.

As described in the previous section, a port type object (ptob) is a structure of type `scm_ptob_descriptor`. A ptob is created by calling `scm_make_port_type`.

`scm_t_bits scm_make_port_type` (*char *name, int* (**fill_input*) (*SCM port*), *void* (**write*) (*SCM port, const void *data, size_t size*)) [Function]

Return a new port type object. The *name*, *fill_input* and *write* parameters are initial values for those port type fields, as described below. The other fields are initialized with default values and can be changed later.

All of the elements of the ptob, apart from `name`, are procedures which collectively implement the port behaviour. 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 is the only element of `scm_ptob_descriptor` which is not a procedure. Set via the first argument to `scm_make_port_type`.

`mark` Called during garbage collection to mark any SCM objects that a port object may contain. It doesn't need to be set unless the port has `SCM` components. Set using

`void scm_set_port_mark` (*scm_t_bits tc, SCM* (**mark*) (*SCM port*)) [Function]

`free` Called when the port is collected during gc. It should free any resources used by the port. Set using

`void scm_set_port_free` (*scm_t_bits tc, size_t* (**free*) (*SCM port*)) [Function]

`print` Called when `write` is called on the port object, to print a port description. E.g., for an fport it may produce something like: `#<input: /etc/passwd 3>`. Set using

`void scm_set_port_print` (*scm_t_bits tc, int* (**print*) (*SCM port, SCM dest_port, scm_print_state *pstate*)) [Function]

The first argument *port* is the object being printed, the second argument *dest_port* is where its description should go.

`equalp` Not used at present. Set using

`void scm_set_port_equalp` (*scm_t_bits tc, SCM* (**equalp*) (*SCM, SCM*)) [Function]

`close` Called when the port is closed, unless it was collected during gc. It should free any resources used by the port. Set using

`void scm_set_port_close` (*scm_t_bits tc, int* (**close*) (*SCM port*)) [Function]

`write` Accept data which is to be written using the port. The port implementation may choose to buffer the data instead of processing it directly. Set via the third argument to `scm_make_port_type`.

`flush` Complete the processing of buffered output data. Reset the value of `rw_active` to `SCM_PORT_NEITHER`. Set using

`void scm_set_port_flush` (*scm_t_bits tc, void* (**flush*) (*SCM port*)) [Function]

`end_input`
Perform any synchronization required when switching from input to output on the port. Reset the value of `rw_active` to `SCM_PORT_NEITHER`. Set using

`void scm_set_port_end_input` (*scm_t_bits tc, void* (**end_input*) (*SCM port, int offset*)) [Function]

`fill_input`
Read new data into the read buffer and return the first character. It can be assumed that the read buffer is empty when this procedure is called. Set via the second argument to `scm_make_port_type`.

`input_waiting`
Return a lower bound on the number of bytes that could be read from the port without blocking. It can be assumed that the current state of `rw_active` is `SCM_PORT_NEITHER`. Set using

`void scm_set_port_input_waiting` (*scm_t_bits tc, int* (**input_waiting*) (*SCM port*)) [Function]

`seek` Set the current position of the port. The procedure can not make any assumptions about the value of `rw_active` when it's called. It can reset the buffers first if desired by using something like:

```
if (pt->rw_active == SCM_PORT_READ)
  scm_end_input (port);
else if (pt->rw_active == SCM_PORT_WRITE)
  ptob->flush (port);
```

However note that this will have the side effect of discarding any data in the unread-char buffer, in addition to any side effects from the `end_input` and `flush` ptob procedures. This is undesirable when seek is called to measure the current position of the port, i.e., `(seek p 0 SEEK_CUR)`. The libguile fport and string port implementations take care to avoid this problem.

The procedure is set using

`void scm_set_port_seek` (*scm_t_bits tc, scm_t_off* (**seek*) (*SCM port, scm_t_off offset, int whence*)) [Function]

`truncate` Truncate the port data to be specified length. It can be assumed that the current state of `rw_active` is `SCM_PORT_NEITHER`. Set using

`void scm_set_port_truncate` (*scm_t_bits tc, void* (**truncate*) (*SCM port, scm_t_off length*)) [Function]

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 man-

ual; an introduction can be found in the Emacs manual (see Section "Syntax of Regular Expressions" in *The GNU Emacs Manual*), or in many general Unix reference books.

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.

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 (`#\nul`) 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.

Patterns and input strings are treated as being in the locale character set if `setlocale` has been called (see Section 7.2.13 [Locales], page 501), and in a multibyte locale this includes treating multi-byte sequences as a single character. (Guile strings are currently merely bytes, though this may change in the future, See Section 6.6.5.13 [Conversion to/from C], page 158.)

`string-match` *pattern str* [*start*] [Scheme Procedure]

Compile the string *pattern* into a regular expression and compare it with *str*. The optional numeric argument *start* specifies the position of *str* at which to begin matching.

`string-match` returns a *match structure* which describes what, if anything, was matched by the regular expression. See Section 6.15.2 [Match Structures], page 340. If *str* does not match *pattern* at all, `string-match` returns `#f`.

Two examples of a match follow. In the first example, the pattern matches the four digits in the match string. In the second, the pattern matches nothing.

```
(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...* [Scheme Procedure]
`scm_make_regexp` (*pat, flaglst*) [C Function]

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` [Variable]

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 341). 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` *rx str* [*start* [*flags*]] [Scheme Procedure]
`scm_regexp_exec` (*rx, str, start, flags*) [C Function]

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 124) 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,

```
(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.

```
(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,

```
(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,

```
(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.

```
(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.

`regexp-match?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a match structure returned by a previous call to `regexp-exec`, or `#f` otherwise.

`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`.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:substring s)
⇒ "2002"

;; match starting at offset 6 in the string
(match:substring
  (string-match "[0-9][0-9][0-9][0-9]" "blah987654" 6))
⇒ "7654"
```

`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:

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:start s)
⇒ 4
```

`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").

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:end s)
⇒ 8
```

`match:prefix` *match* [Scheme Procedure]

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* [Scheme Procedure]

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* [Scheme Procedure]

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:string` *match* [Scheme Procedure]

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\`' followed by any number of alphabetic characters. The regular expression '`\\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* [Scheme Procedure]

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:

```
(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:

```
(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 (`http://code.google.com/p/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 Reading and Evaluating Scheme Code

This chapter describes Guile functions that are concerned with reading, loading, evaluating, and compiling Scheme code at run time.

6.17.1 Scheme Syntax: Standard and Guile Extensions

6.17.1.1 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,

```
(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.

```
(max 1 2 3) ⇒ 3

(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 248, and see Section 6.10.2 [Syntax Rules], page 249).

constant Number, string, character and boolean constants evaluate "to themselves", so can appear as literals.

```
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)`
`'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,

```
'x                    ⇒ x
'(1 2 3)              ⇒ (1 2 3)
'#(1 (2 3) 4)         ⇒ #(1 (2 3) 4)
(quote x)             ⇒ x
(quote (1 2 3))       ⇒ (1 2 3)
(quote #(1 (2 3) 4)) ⇒ #(1 (2 3) 4)
```

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)`
`` `data `` Backquote quasi-quotation 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 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 (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.

6.17.1.2 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.

6.17.1.3 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 `!#`, which must appear on a line of their own. These comments are compatible with the block comments in the Scheme Shell '`scsh`' (see Section 7.17 [The Scheme shell (scsh)], page 661). 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 40).

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`):

```
(+  #| this is a #| nested |# block comment |# 2)
⇒ 3
```

For backward compatibility, this syntax can be overridden with `read-hash-extend` (see Section 6.17.1.6 [Reader Extensions], page 346).

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 6.17.8 [Character Encoding of Source Files], page 355.

6.17.1.4 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
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 6.17.2 [Scheme Read], page 346.

```
(read-enable 'case-insensitive)
```

Note that this is seldom a problem, because Scheme programmers tend not to use uppercase letters in their identifiers anyway.

6.17.1.5 Keyword Syntax

6.17.1.6 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.

6.17.2 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 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.
```

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* [Scheme Syntax]

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).

6.17.3 Writing Scheme Values

Any scheme value may be written to a port. Not all values may be read back in (see Section 6.17.2 [Scheme Read], page 346), however.

`write` *obj* [*port*] [Scheme Procedure]

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 6.17.2 [Scheme Read], page 346). Strings are printed in double quotes, with escapes if necessary, and characters are printed in '`#\`' notation.

`display` *obj* [*port*] [Scheme Procedure]

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*] [Scheme Procedure]

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'.
```

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.

6.17.4 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 6.19.12 [Environments], page 381, 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 6.23 [Other Languages], page 417). 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:

```
(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 6.19.8 [Module System Reflection], page 375) 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.

`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.

`SCM scm_c_eval_string` (*const char *string*) [C Function]

`scm_eval_string`, but taking a C string in locale encoding instead of an `SCM`.

`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.

`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.

`scm_call` (*proc, ...*) [C Function]

Call *proc* with any number of arguments. The argument list must be terminated by `SCM_UNDEFINED`. For example:

```
scm_call (scm_c_public_ref ("guile", "+"),
          scm_from_int (1),
          scm_from_int (2),
          SCM_UNDEFINED);
```

`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`.

`apply:nconc2last` *lst* [Scheme Procedure]
`scm_nconc2last` (*lst*) [C Function]

lst should be a list (*arg1* ... *argN arglst*), with *arglst* being a list. This function returns a list comprising *arg1* to *argN* plus the elements of *arglst*. *lst* is modified to

form the return. *arglst* is not modified, though the return does share structure with it.

This operation collects up the arguments from a list which is `apply` style parameters.

`primitive-eval` *exp* [Scheme Procedure]
`scm_primitive_eval` (*exp*) [C Function]

Evaluate *exp* in the top-level environment specified by the current module.

6.17.5 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 10.4 [Compiling to the Virtual Machine], page 757. For information on the virtual machine, see Section 10.3 [A Virtual Machine for Guile], page 738.

The command-line interface to Guile's compiler is the `guild compile` command:

`guild compile` ['`option`'...] *file*... [Command]

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).

'-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`, `unbound-variable`, `arity-mismatch`, and `format`.

'-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, `objcode` is assumed.

'-T *target*'
'--target=*target*'
Produce bytecode for *target* instead of *%host-type* (see Section 6.22.1 [Build Config], page 412). 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 6.17.8 [Character Encoding of Source Files], page 355).

The compiler can also be invoked directly by Scheme code using the procedures below:

`compile` *exp* [*env*=#f] [*from*=(*current-language*)] [*to*=*value*] [*opts*=()] [Scheme Procedure]

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 10.4 [Compiling to the Virtual Machine], page 757.

`compile-file` *file* [*output-file*=#f] [*from*=(*current-language*)] [*to*='*objcode*] [*env*=(*default-environment from*)] [*opts*='()] [*canonicalization* '*relative*] [Scheme Procedure]

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 10.4 [Compiling to the Virtual Machine], page 757, 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* [Scheme Procedure]

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` [Scheme Variable]

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`'.

6.17.6 Loading Scheme Code from File

`load` *filename* [*reader*] [Scheme Procedure]

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 6.17.2 [Scheme Read], page 346).

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* [Scheme Procedure]

Load the compiled file named *filename*.

Compiling a source file (see Section 6.17 [Read/Load/Eval/Compile], page 343) and then calling `load-compiled` on the resulting file is equivalent to calling `load` on the source file.

`primitive-load` *filename* [Scheme Procedure]
`scm_primitive_load` (*filename*) [C Function]

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*) [C Function]

`scm_primitive_load`, but taking a C string instead of an `SCM`.

`current-reader` [Variable]

`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 6.21.7 [Fluids and Dynamic States], page 405).

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` [Variable]

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 ...\n" filename)))
(load-from-path "foo.scm")
⊣ Loading /usr/local/share/guile/site/foo.scm ...
```

`current-load-port` [Scheme Procedure]
`scm_current_load_port` () [C Function]

Return the current-load-port. The load port is used internally by `primitive-load`.

6.17.7 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` [Variable]

List of directories which should be searched for Scheme modules and libraries. `%load-path` is initialized when Guile starts up to `(list (%site-dir) (%library-dir) (%package-data-dir))`, prepended with the contents of the `GUILE_LOAD_PATH` environment variable, if it is set. See Section 6.22.1 [Build Config], page 412, for more on `%site-dir` and related procedures.

`load-from-path` *filename* [Scheme Procedure]

Similar to `load`, but searches for *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 `add-to-load-path`.

`add-to-load-path` *dir* [Scheme Syntax]

Add *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:

```
(add-to-load-path (dirname (current-filename)))
```

It's better to use `add-to-load-path` than to modify `%load-path` directly, because `add-to-load-path` takes care of modifying the path both at compile-time and at run-time.

`primitive-load-path` *filename* [*exception-on-not-found*] [Scheme Procedure]
`scm_primitive_load_path` (*filename*) [C Function]

Search `%load-path` for the file named *filename* and load it into the top-level environment. If *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.

By default or if *exception-on-not-found* is true, an exception is raised if *filename* is not found. If *exception-on-not-found* is `#f` and *filename* is not found, no exception is raised and `#f` is returned. 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.

`%search-load-path` *filename* [Scheme Procedure]
`scm_sys_search_load_path` (*filename*) [C Function]

Search `%load-path` for the file named *filename*, which must be readable by the current user. If *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.

`%load-extensions` [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.

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/2.0/ccache/ice-9/popen.go`, and use it instead of `/usr/share/guile/2.0/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 54, for more on how to correctly install site packages. See Section 6.19.4 [Modules and the File System], page 370, for more on the relationship between load paths and modules. See Section 6.17.5 [Compilation], page 350, 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.

`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.

6.17.8 Character Encoding of Source Files

Scheme source code files are usually encoded in ASCII, but, the built-in reader can interpret other character encodings. The procedure `primitive-load`, and by extension the functions that call it, such as `load`, first scan the top 500 characters of the file for a coding declaration.

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 a block `#!` comment.

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 304). 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 304).

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.

`file-encoding` *port* [Scheme Procedure]
`scm_file_encoding` (*port*) [C Function]

Scan the port for an Emacs-like character coding declaration near the top of the contents of a port with random-accessible contents (see Section "Recognize Coding" in *The GNU Emacs Reference Manual*). The coding declaration is of the form `coding: XXXXX` and must appear in a Scheme comment. Return a string containing the character encoding of the file if a declaration was found, or `#f` otherwise. The port is rewound.

6.17.9 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.

`delay` *expr* [syntax]

Return a promise object which holds the given *expr* expression, ready to be evaluated by a later `force`.

`promise?` *obj* [Scheme Procedure]
`scm_promise_p` (*obj*) [C Function]

Return true if *obj* is a promise.

`force` *p* [Scheme Procedure]
`scm_force` (*p*) [C Function]

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.

6.17.10 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` [syntax]

Captures and returns a lexical environment for use with `local-eval` or `local-compile`.

`local-eval` *exp env* [Scheme Procedure]
`scm_local_eval` (*exp, env*) [C Function]
`local-compile` *exp env* [*opts*=()] [Scheme Procedure]

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.

6.17.11 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* [Scheme Syntax]

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 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 263.)

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* [Scheme Syntax]

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 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 65 for a general discussion of how garbage collection relates to using Guile from C.

6.18.1 Function related to Garbage Collection

`gc` [Scheme Procedure]
`scm_gc ()` [C Function]

Scans all of SCM objects and reclaims for further use those that are no longer accessible. You normally don't need to call this function explicitly. It is called automatically when appropriate.

`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`/`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 unprotect 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[10].

`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]
`void scm_remember_upto_here_2` (*SCM obj1, SCM obj2*) [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.

6.18.2 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 smob) 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 can be

[10] In Guile up to version 1.8, C global variables were not scanned by the garbage collector; hence, `scm_gc_protect_object` was the only way in C to prevent a Scheme object from being freed.

freed with `scm_gc_free`; however, this is not strictly needed: 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[11].

Memory allocated with `scm_gc_malloc` is scanned 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[12]. Conversely, memory allocated with `scm_gc_malloc_pointerless` is assumed to be "pointer-less" and is not scanned.

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 298.

`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. When *mem* is `NULL`, this function behaves like `scm_malloc` and allocates a new block of size *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 `scm_gc_register_allocation`.

`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]

[11] In Guile up to version 1.8, memory allocated with `scm_gc_malloc` *had* to be freed with `scm_gc_free`.

[12] In Guile up to 1.8, memory allocated with `scm_gc_malloc` was *not* scanned. 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 233)

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]

Allocate *size* bytes of automatically-managed memory. The memory is automatically freed when no longer referenced from any live memory block.

Memory allocated with scm_gc_malloc or scm_gc_calloc is scanned for pointers. Memory allocated by scm_gc_malloc_pointerless is not scanned.

The scm_gc_realloc call preserves the "pointerlessness" of the memory area pointed to by *mem*. Note that you need to pass the old size of a reallocated memory block as well. See below for a motivation.

void scm_gc_free (*void *mem*, *size_t size*, *const char *what*) [C Function]

Explicitly free the memory block pointed to by *mem*, which was previously allocated by one of the above scm_gc functions.

Note that you need to explicitly pass the *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, *size* is always ignored.

void scm_gc_register_allocation (*size_t size*) [C Function]

Informs the garbage collector that *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 SMOBs.

void scm_dynwind_free (*void *mem*) [C Function]

Equivalent to scm_dynwind_unwind_handler (free, *mem*, SCM_F_WIND_EXPLICITLY). That is, the memory block at *mem* will be freed (using free from the C library) when the current dynwind is left.

malloc-stats [Scheme Procedure]

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.

6.18.2.1 Upgrading from scm_must_malloc et al.

Version 1.6 of Guile and earlier did not have the functions from the previous section. In their place, it had the functions scm_must_malloc, scm_must_realloc and scm_must_free. This section explains why we want you to stop using them, and how to do this.

The functions scm_must_malloc and scm_must_realloc behaved like scm_gc_malloc and scm_gc_realloc do now, respectively. They would inform the GC about the newly

allocated memory via the internal equivalent of `scm_gc_register_allocation`. However, `scm_must_free` did not unregister the memory it was about to free. The usual way to unregister memory was to return its size from a smob free function.

This disconnectedness of the actual freeing of memory and reporting this to the GC proved to be bad in practice. It was easy to make mistakes and report the wrong size because allocating and freeing was not done with symmetric code, and because it is cumbersome to compute the total size of nested data structures that were freed with multiple calls to `scm_must_free`. Additionally, there was no equivalent to `scm_malloc`, and it was tempting to just use `scm_must_malloc` and never to tell the GC that the memory has been freed.

The effect was that the internal statistics kept by the GC drifted out of sync with reality and could even overflow in long running programs. When this happened, the result was a dramatic increase in (senseless) GC activity which would effectively stop the program dead.

The functions `scm_done_malloc` and `scm_done_free` were introduced to help restore balance to the force, but existing bugs did not magically disappear, of course.

Therefore we decided to force everybody to review their code by deprecating the existing functions and introducing new ones in their place that are hopefully easier to use correctly.

For every use of `scm_must_malloc` you need to decide whether to use `scm_malloc` or `scm_gc_malloc` in its place. When the memory block is not part of a smob or some other Scheme object whose lifetime is ultimately managed by the garbage collector, use `scm_malloc` and `free`. When it is part of a smob, use `scm_gc_malloc` and change the smob free function to use `scm_gc_free` instead of `scm_must_free` or `free` and make it return zero.

The important thing is to always pair `scm_malloc` with `free`; and to always pair `scm_gc_malloc` with `scm_gc_free`.

The same reasoning applies to `scm_must_realloc` and `scm_realloc` versus `scm_gc_realloc`.

6.18.3 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 267)

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.

Hashtables 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.

6.18.3.1 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. (see Section 6.7.13 [Hash Tables], page 228)

`weak-key-hash-table?` *obj* [Scheme Procedure]
`weak-value-hash-table?` *obj* [Scheme Procedure]
`doubly-weak-hash-table?` *obj* [Scheme Procedure]
`scm_weak_key_hash_table_p` (*obj*) [C Function]
`scm_weak_value_hash_table_p` (*obj*) [C Function]
`scm_doubly_weak_hash_table_p` (*obj*) [C Function]

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.

6.18.3.2 Weak vectors

Weak vectors are mainly useful in Guile's implementation of weak hash tables.

`make-weak-vector` *size* [*fill*] [Scheme Procedure]
`scm_make_weak_vector` (*size*, *fill*) [C Function]

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* ... [Scheme Procedure]
`list->weak-vector` *l* [Scheme Procedure]
`scm_weak_vector` (*l*) [C Function]

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* [Scheme Procedure]
`scm_weak_vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a weak vector. Note that all weak hashes are also weak vectors.

6.18.4 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.

See R. Kent Dybvig, Carl Bruggeman, and David Eby (1993) "Guardians in a Generation-Based Garbage Collector". ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1993.

`make-guardian` [Scheme Procedure]
`scm_make_guardian ()` [C Function]

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.

6.19 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.

6.19.1 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 6.19.3 [Creating Guile Modules], page 367).

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 6.19.2 [Using Guile Modules], page 365).

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.

6.19.2 Using Guile Modules

To use a Guile module is to access either its public interface or a custom interface (see Section 6.19.1 [General Information about Modules], page 365). 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 6.19.4 [Modules and the File System], page 370).

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:

```
(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 6.19.10 [Included Guile Modules], page 379).

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:)))
```

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:

```
(ice-9 popen) sees:          current module sees:
open-pipe                    unixy:pipe-open
close-pipe                   unixy:close-pipe
```

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 yet 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] [#: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; and *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 `(ORIG . SEEN)`, where *orig* is the name in the used module and *seen* is the name in the using module. Note that *seen* is also passed through *renamer*.

The `#:select` and `#:renamer` clauses are optional. If both are omitted, the returned interface has no bindings. If the `#:select` clause is omitted, *renamer* operates on the used module's public interface.

In addition to the above, *spec* can also include a `#:version` clause, of the form:

```
#:version VERSION-SPEC
```

where *version-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 *version-spec*. See Section 6.19.5 [R6RS Version References], page 371, for more on version references.

If the module name is not resolvable, `use-modules` will signal an error.

`@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must have been exported by the module.

`@@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *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.

6.19.3 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 `define-module` form at the beginning.
- Export all bindings which should be in the public interface, either by using `define-public` or `export` (both documented below).

`define-module` *module-name option* . . . [syntax]

module-name is a list of one or more symbols.

```
(define-module (ice-9 popen))
```

`define-module` makes this module available to Guile programs under the given *module-name*.

option . . . are keyword/value pairs which specify more about the defined module. The recognized options and their meaning are shown in the following table.

`#:use-module` *interface-specification*

Equivalent to a `(use-modules` *interface-specification*`)` (see Section 6.19.2 [Using Guile Modules], page 365).

`#:autoload` *`module symbol-list`*

Load *module* when any of *symbol-list* are accessed. For example,

```
(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 6.19.2 [Using Guile Modules], page 365), 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 568) which is not compatible with the core `current-time` function (see Section 7.2.5 [Time], page 468). 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 9.6.3 [Merging Generics], page 694, 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.

`export` *variable* ... [syntax]

Add all *variable*s (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.

`define-public` ... [syntax]

Equivalent to `(begin (define foo ...) (export foo))`.

`re-export` *variable* ... [syntax]

Add all *variable*s (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.

`export!` *variable* ... [syntax]

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.

6.19.4 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 6.17.7 [Load Paths], page 353). 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 6.17.7 [Load Paths], page 353, 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 6.17.5 [Compilation], page 350, 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 54.

6.19.5 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 Revised^6 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)))
```

6.19.6 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))
  (import (otherlib (3)))
  (export mybinding))
```

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 ...*) *body ...* [Scheme Syntax]

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 6.19.3 [Creating Guile Modules], page 367).

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 6.19.5 [R6RS Version References], page 371), 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.

6.19.7 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 drawn no distinction between — a name that is unbound and a name that is bound to a variable whose value is undefined. We will try to be clear about the difference and explain any confusion where it is unavoidable.)

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`.

`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` iff *var* is bound to a value. 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` iff *obj* is a variable object, else return `#f`.

6.19.8 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` [Scheme Procedure]
`scm_current_module ()` [C Function]
Return the current module object.

`set-current-module` *module* [Scheme Procedure]
`scm_set_current_module` (*module*) [C Function]
Set the current module to *module* and return the previous current module.

`save-module-excursion` *thunk* [Scheme Procedure]
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 298).

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*] [Scheme Procedure]
`scm_resolve_module` (*name*) [C Function]
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 6.19.5 [R6RS Version References], page 371). The name is a list of symbols.

`resolve-interface` *name* [*#:select=#f*] [*#:hide='()*] [*#:select=()*] [*#:prefix=#f*] [*#:renamer*] [*#:version=#f*] [Scheme Procedure]
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* [Scheme Procedure]
Return a list of the interfaces used by *module*.

`module-use!` *module interface* [Scheme Procedure]
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* [Scheme Procedure]
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 6.19.9 [Accessing Modules from C], page 376.

`module-variable` *module name* [Scheme Procedure]
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.

6.19.9 Accessing Modules from C

The last sections have described how modules are used in Scheme code, which is the recommended 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`) [C Function]
`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`) [C Function]
`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.

`SCM scm_public_lookup` (*SCM* `module_name`, *SCM* `name`) [C Function]
`SCM scm_c_public_lookup` (*const char* *`module_name`, *const char* *`name`) [C Function]
`SCM scm_private_lookup` (*SCM* `module_name`, *SCM* `name`) [C Function]
`SCM scm_c_private_lookup` (*const char* *`module_name`, *const char* *`name`) [C Function]

Like `scm_public_variable` or `scm_private_variable`, but if the *name* is not bound in the module, signals an error. Returns a variable, always.

```
SCM my_eval_string (SCM str)
{
  static SCM eval_string_var = SCM_BOOL_F;

  if (scm_is_false (eval_string_var))
    eval_string_var =
      scm_c_public_lookup ("ice-9 eval-string", "eval-string");

  return scm_call_1 (scm_variable_ref (eval_string_var), str);
}
```

`SCM scm_public_ref` (*SCM* `module_name`, *SCM* `name`) [C Function]
`SCM scm_c_public_ref` (*const char* *`module_name`, *const char* *`name`) [C Function]
`SCM scm_private_ref` (*SCM* `module_name`, *SCM* `name`) [C Function]
`SCM scm_c_private_ref` (*const char* *`module_name`, *const char* *`name`) [C Function]

Like `scm_public_lookup` or `scm_private_lookup`, but additionally dereferences the variable. If the variable object is unbound, signals an error. Returns the value bound to *name* in *module_name*.

In addition, there are a number of other lookup-related procedures. We suggest that you use the `scm_public_` and `scm_private_` family of procedures instead, if possible.

`SCM scm_c_lookup` (*const char* *`name`) [C Function]

Return the variable bound to the symbol indicated by *name* in the current module. If there is no such binding or the symbol is not bound to a variable, signal an error.

SCM scm_lookup (*SCM name*) [C Function]
Like scm_c_lookup, but the symbol is specified directly.

SCM scm_c_module_lookup (*SCM module, const char *name*) [C Function]
SCM scm_module_lookup (*SCM module, SCM name*) [C Function]
Like scm_c_lookup and 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`.

`SCM 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`.

6.19.10 Included Guile Modules

Some modules are included in the Guile distribution; here are references to the entries in this manual which describe them in more detail:

boot-9 boot-9 is Guile's initialization module, and it is always loaded when Guile starts up.

(ice-9 expect)
Actions based on matching input from a port (see Section 7.15 [Expect], page 653).

(ice-9 format)
Formatted output in the style of Common Lisp (see Section 7.10 [Formatted Output], page 633).

(ice-9 ftw) File tree walker (see Section 7.11 [File Tree Walk], page 644).

(ice-9 getopt-long)
Command line option processing (see Section 7.4 [getopt-long], page 528).

(ice-9 history)
Refer to previous interactive expressions (see Section 4.4.3 [Value History], page 46).

(ice-9 popen)
Pipes to and from child processes (see Section 7.2.10 [Pipes], page 482).

(ice-9 pretty-print)
Nicely formatted output of Scheme expressions and objects (see Section 7.9 [Pretty Printing], page 632).

(ice-9 q) First-in first-out queues (see Section 7.12 [Queues], page 649).

(ice-9 rdelim)
Line- and character-delimited input (see Section 6.14.6 [Line/Delimited], page 310).

(ice-9 readline)
`readline` interactive command line editing (see Section 7.8 [Readline Support], page 628).

(ice-9 receive)
Multiple-value handling with `receive` (see Section 6.13.7 [Multiple Values], page 290).

(ice-9 regex)
Regular expression matching (see Section 6.15 [Regular Expressions], page 335).

(ice-9 rw) Block string input/output (see Section 6.14.7 [Block Reading and Writing], page 312).

(ice-9 streams)
Sequence of values calculated on-demand (see Section 7.13 [Streams], page 650).

(ice-9 syncase)
R5RS `syntax-rules` macro system (see Section 6.10.2 [Syntax Rules], page 249).

(ice-9 threads)
Guile's support for multi threaded execution (see Section 6.21 [Scheduling], page 396).

(ice-9 documentation)
Online documentation (REFFIXME).

(srfi srfi-1)
A library providing a lot of useful list and pair processing procedures (see Section 7.5.3 [SRFI-1], page 535).

(srfi srfi-2)
Support for `and-let*` (see Section 7.5.4 [SRFI-2], page 548).

(srfi srfi-4)
Support for homogeneous numeric vectors (see Section 7.5.5 [SRFI-4], page 549).

(srfi srfi-6)
Support for some additional string port procedures (see Section 7.5.6 [SRFI-6], page 558).

(srfi srfi-8)
Multiple-value handling with `receive` (see Section 7.5.7 [SRFI-8], page 558).

(srfi srfi-9)
Record definition with `define-record-type` (see Section 7.5.8 [SRFI-9], page 558).

(srfi srfi-10)
Read hash extension `#,()` (see Section 7.5.9 [SRFI-10], page 560).

(srfi srfi-11)
Multiple-value handling with `let-values` and `let*-values` (see Section 7.5.10 [SRFI-11], page 562).

(srfi srfi-13)
String library (see Section 7.5.11 [SRFI-13], page 562).

(srfi srfi-14)
Character-set library (see Section 7.5.12 [SRFI-14], page 562).

(srfi srfi-16)
`case-lambda` procedures of variable arity (see Section 7.5.13 [SRFI-16], page 562).

(srfi srfi-17)
Getter-with-setter support (see Section 7.5.14 [SRFI-17], page 562).

(srfi srfi-19)
Time/Date library (see Section 7.5.16 [SRFI-19], page 568).

(srfi srfi-26)
Convenient syntax for partial application (see Section 7.5.18 [SRFI-26], page 575)

(srfi srfi-31)
`rec` convenient recursive expressions (see Section 7.5.21 [SRFI-31], page 578)

(ice-9 slib)
This module contains hooks for using Aubrey Jaffer's portable Scheme library SLIB from Guile (see Section 7.1 [SLIB], page 451).

6.19.11 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.

6.19.12 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 6.17.4 [Fly Evaluation], page 348). Three procedures are defined to return environments: `scheme-report-environment`, `null-environment` and `interaction-environment` (see Section 6.17.4 [Fly Evaluation], page 348).

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.

6.20 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.

6.20.1 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 his 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*.[13]

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*] [Scheme Procedure]
`scm_dynamic_link` (*library*) [C Function]

Find the shared library denoted by *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, *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 '`/usr/lib`' and '`/usr/local/lib`'.

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.

`dynamic-object?` *obj* [Scheme Procedure]
`scm_dynamic_object_p` (*obj*) [C Function]

Return `#t` if *obj* is a dynamic library handle, or `#f` otherwise.

`dynamic-unlink` *dobj* [Scheme Procedure]
`scm_dynamic_unlink` (*dobj*) [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.

```
(define libgl-obj (dynamic-link "libGL"))
libgl-obj
⇒ #<dynamic-object "libGL">
(dynamic-unlink libGL-obj)
libGL-obj
⇒ #<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.

[13] 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 *dlopening*, as suggested by Gordon Matzigkeit in his libtool documentation.

6.20.2 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.

`dynamic-func` *name dobj* [Scheme Procedure]
`scm_dynamic_func` (*name, dobj*) [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`.

`dynamic-call` *func dobj* [Scheme Procedure]
`scm_dynamic_call` (*func, dobj*) [C Function]

Call the C function indicated by *func* and *dobj*. The function is passed no arguments and its return value is ignored. When *function* is something returned by `dynamic-func`, call that function and ignore *dobj*. When *func* is a string , look it up in *dynobj*; this is equivalent to

```
(dynamic-call (dynamic-func func dobj) #f)
```

Interrupts are deferred while the C function is executing (with `SCM_DEFER_INTS/SCM_ALLOW_INTS`).

`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 '`libX11`' with `dynamic-link` and then construct a beautiful graphical user interface just by using `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 '`libX11`' wrapper in Scheme via the dynamic FFI. See Section 6.20.6 [Dynamic FFI], page 393, 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.

`load-extension` *lib init* [Scheme Procedure]
`scm_load_extension` (*lib, init*) [C Function]

Load and initialize the extension designated by LIB and INIT. When there is no pre-registered function for LIB/INIT, this is equivalent to

```
(dynamic-call INIT (dynamic-link LIB))
```

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 `dynamic-link`, *lib* should not contain any suffix such as `.so` (see Section 6.20.1 [Foreign Libraries], page 382). 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 '`libguile-bla-blum`' for a extension related to a module `(bla blum)`.

The normal way for a 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,

```
(define-module (bla blum))

(load-extension "libguile-bla-blum" "bla_init_blum")
```

6.20.3 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.

```
#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"))
(dynamic-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, `j0` is contained in the root module, where all the other Guile primitives like `display`, etc live. In general, a primitive is put into whatever module is the *current module* at the time `scm_c_define_gsubr` is called.

A compiled module should have a specially named *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 `init_math_bessel` with the following code in '`bessel.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: '`scm_init_`', followed by the name of the module where the individual hierarchical components are concatenated with underscores, followed by '`_module`'.

After '`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> (load-extension "./libbessel.so" "scm_init_math_bessel_module")
guile> (use-modules (math bessel))
guile> (j0 2)
0.223890779141236
guile> (apropos "j0")
⊣ (math bessel): j0      #<primitive-procedure j0>
```

That's it!

6.20.4 Modules and Extensions

The new primitives that you add to Guile with `scm_c_define_gsubr` (see Section 6.9.2 [Primitive Procedures], page 237) or with any of the other mechanisms are placed into the

module that is current when the `scm_c_define_gsubr` is executed. Extensions loaded from the REPL, for example, will be placed into the `(guile-user)` module, if the REPL module was not changed.

To define C primitives within a specific module, the simplest way is:

```
(define-module (foo bar))
(load-extension "foobar-c-code" "foo_bar_init")
```

When loaded with `(use-modules (foo bar))`, the `load-extension` call looks for the '`foobar-c-code.so`' (etc) object file in Guile's `extensiondir`, which is usually a subdirectory of the `libdir`. For example, if your libdir is '`/usr/lib`', the `extensiondir` for the Guile 2.0.*x* series will be '`/usr/lib/guile/2.0/`'.

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.

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`'

```
(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*),

```
foo.scm: foo.scm.in
        sed 's|XXextensiondirXX|$(libdir)/guile/2.0|' <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.

```
(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 6.24 [Internationalization], page 420).

It will be noted all of the above requires that the Scheme code to be found in `%load-path` (see Section 6.17.7 [Load Paths], page 353). 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.

6.20.5 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.

6.20.5.1 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]
`size_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.

6.20.5.2 Foreign Variables

Pointers to variables in the current address space may be looked up dynamically using `dynamic-pointer`.

`dynamic-pointer` *name dobj* [Scheme Procedure]
`scm_dynamic_pointer` (*name, dobj*) [C Function]

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* [Scheme Procedure]
`scm_pointer_address` (*pointer*) [C Function]

Return the numerical value of *pointer*.

```
(pointer-address numptob)
⇒ 139984413364296 ; YMMV
```

`make-pointer` *address* [*finalizer*] [Scheme Procedure]

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* [Scheme Procedure]

Return `#t` if *obj* is a pointer object, `#f` otherwise.

`%null-pointer` [Scheme Variable]

A foreign pointer whose value is 0.

`null-pointer?` *pointer* [Scheme Procedure]

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* [Scheme Procedure]

Return a foreign pointer object with the `object-address` of *scm*.

`pointer->scm` *pointer* [Scheme Procedure]

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**)) [C Function]

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*) [C Function]

Unpack the pointer value from a pointer object.

6.20.5.3 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 6.6.6 [Bytevectors], page 162). The `(rnrs bytevector)` 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*]] [Scheme Procedure]
`scm_pointer_to_bytevector` (*pointer, len, offset, uvec_type*) [C Function]

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 `uniform-vector-element-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*] [Scheme Procedure]
`scm_bytevector_to_pointer` (*bv, offset*) [C Function]

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* [Scheme Procedure]

Assuming *pointer* points to a memory region that holds a pointer, return this pointer.

`string->pointer` *string* [*encoding*] [Scheme Procedure]

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*] [Scheme Procedure]

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` *type-name pred wrap unwrap print* [Scheme Syntax]

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.

6.20.5.4 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* [Scheme Procedure]

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; }`:

```
(parse-c-struct (make-c-struct (list int64 uint8)
                               (list 300 43))
                (list int64 uint8))
⇒ (300 43)
```

As yet, Guile only has convenience routines to support conventionally-packed structs. But given the `bytevector->foreign` and `foreign->bytevector` routines, one can create and parse tightly packed structs and unions by hand. See the code for `(system foreign)` for details.

6.20.6 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]
`scm_pointer_to_procedure` (*return_type, func_ptr, arg_types*) [C Procedure]

Make a foreign function.

Given the foreign void pointer *func_ptr*, its argument and return types *arg_types* and *return_type*, return a procedure that will pass arguments to the foreign function and return appropriate values.

arg_types should be a list of foreign types. `return_type` should be a foreign type. See Section 6.20.5.1 [Foreign Types], page 388, for more information on foreign types.

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, `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 `*`, indicating a pointer. For example, the following code makes `memcpy` available to Scheme:

```
(define memcpy
  (let ((this (dynamic-link)))
    (pointer->procedure '*
                        (dynamic-func "memcpy" this)
                        (list '* '* size_t))))
```

To invoke `memcpy`, one must pass it foreign pointers:

```
(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)
```

One may also pass structs as values, passing structs as foreign pointers. See Section 6.20.5.4 [Foreign Structs], page 392, 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.

```
;; struct timeval {
;;      time_t      tv_sec;     /* seconds */
;;      suseconds_t tv_usec;    /* microseconds */
;; };
;; assuming fields are of type "long"

(define gettimeofday
  (let ((f (pointer->procedure
            int
            (dynamic-func "gettimeofday" (dynamic-link))
            (list '* '*)))
        (tv-type (list long long)))
    (lambda ()
      (let* ((timeval (make-c-struct tv-type (list 0 0)))
             (ret (f timeval %null-pointer)))
```

```
            (if (zero? ret)
                (apply values (parse-c-struct timeval tv-type))
                (error "gettimeofday returned an error" ret))))))

(gettimeofday)
⇒ 1270587589
⇒ 499553
```

As you can see, this interface to foreign functions is at a very low, somewhat dangerous level[14].

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 C function.

`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*.

As an example, here's how the C library's `qsort` array sorting function can be made accessible to Scheme (see Section "Array Sort Function" in *The GNU C Library Reference Manual*):

```
(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)))
```

[14] 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.

6.21 Threads, Mutexes, Asyncs and Dynamic Roots

6.21.1 Arbiters

Arbiters are synchronization objects, they can be used by threads to control access to a shared resource. An arbiter can be locked to indicate a resource is in use, and unlocked when done.

An arbiter is like a light-weight mutex (see Section 6.21.4 [Mutexes and Condition Variables], page 400). It uses less memory and may be faster, but there's no way for a thread to block waiting on an arbiter, it can only test and get the status returned.

`make-arbiter` *name* [Scheme Procedure]
`scm_make_arbiter` (*name*) [C Function]

Return an object of type arbiter and name *name*. Its state is initially unlocked. Arbiters are a way to achieve process synchronization.

`try-arbiter` *arb* [Scheme Procedure]
`scm_try_arbiter` (*arb*) [C Function]

If *arb* is unlocked, then lock it and return `#t`. If *arb* is already locked, then do nothing and return `#f`.

`release-arbiter` *arb* [Scheme Procedure]
`scm_release_arbiter` (*arb*) [C Function]

If *arb* is locked, then unlock it and return `#t`. If *arb* is already unlocked, then do nothing and return `#f`.

Typical usage is for the thread which locked an arbiter to later release it, but that's not required, any thread can release it.

6.21.2 Asyncs

Asyncs are a means of deferring the execution of Scheme code until it is safe to do so.

Guile provides two kinds of asyncs that share the basic concept but are otherwise quite different: system asyncs and user asyncs. System asyncs are integrated into the core of Guile and are executed automatically when the system is in a state to allow the execution of Scheme code. For example, it is not possible to execute Scheme code in a POSIX signal handler, but such a signal handler can queue a system async to be executed in the near future, when it is safe to do so.

System asyncs can also be queued for threads other than the current one. This way, you can cause threads to asynchronously execute arbitrary code.

User asyncs offer a convenient means of queuing procedures for future execution and triggering this execution. They will not be executed automatically.

6.21.2.1 System asyncs

To cause the future asynchronous execution of a procedure in a given thread, use `system-async-mark`.

Automatic invocation of system asyncs can be temporarily disabled by calling `call-with-blocked-asyncs`. 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 asyncs.

Analogously, the procedure `call-with-unblocked-asyncs` will temporarily decrease the blocking level of the current thread. You can use it when you want to disable asyncs by default and only allow them temporarily.

In addition to the C versions of `call-with-blocked-asyncs` and `call-with-unblocked-asyncs`, C code can use `scm_dynwind_block_asyncs` and `scm_dynwind_unblock_asyncs` inside a *dynamic context* (see Section 6.13.10 [Dynamic Wind], page 298) to block or unblock system asyncs temporarily.

`system-async-mark` *proc* [*thread*] [Scheme Procedure]
`scm_system_async_mark` (*proc*) [C Function]
`scm_system_async_mark_for_thread` (*proc, thread*) [C Function]

Mark *proc* (a procedure with zero arguments) for future execution in *thread*. When *proc* has already been marked 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.

This procedure is not safe to be called from signal handlers. Use `scm_sigaction` or `scm_sigaction_for_thread` to install signal handlers.

`call-with-blocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_blocked_asyncs` (*proc*) [C Function]

Call *proc* and block the execution of system asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void * scm_c_call_with_blocked_asyncs` (*void* * (**proc*) (*void* **data*), *void* **data*) [C Function]

The same but with a C function *proc* instead of a Scheme thunk.

`call-with-unblocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_unblocked_asyncs` (*proc*) [C Function]

Call *proc* and unblock the execution of system asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void * scm_c_call_with_unblocked_asyncs` (*void* *(**proc*) (*void* **data*), *void* **data*) [C Function]

The same but with a C function *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 `scm_dynwind_begin` and `scm_dynwind_end` (see Section 6.13.10 [Dynamic Wind], page 298).

`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 `scm_dynwind_begin` and `scm_dynwind_end` (see Section 6.13.10 [Dynamic Wind], page 298).

6.21.2.2 User asyncs

A user async is a pair of a thunk (a parameterless procedure) and a mark. Setting the mark on a user async will cause the thunk to be executed when the user async is passed to `run-asyncs`. Setting the mark more than once is satisfied by one execution of the thunk.

User asyncs are created with `async`. They are marked with `async-mark`.

`async` *thunk* [Scheme Procedure]
`scm_async` (*thunk*) [C Function]

Create a new user async for the procedure *thunk*.

`async-mark` *a* [Scheme Procedure]
`scm_async_mark` (*a*) [C Function]

Mark the user async *a* for future execution.

`run-asyncs` *list_of_a* [Scheme Procedure]
`scm_run_asyncs` (*list_of_a*) [C Function]

Execute all thunks from the marked asyncs of the list *list_of_a*.

6.21.3 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 6.22.2.1 [Feature Manipulation], page 414).

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 6.21.9 [Futures], page 409).

`all-threads` [Scheme Procedure]
`scm_all_threads ()` [C Function]

Return a list of all threads.

`current-thread` [Scheme Procedure]
`scm_current_thread ()` [C Function]

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*) [C Function]

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* [Scheme Procedure]
`scm_thread_p` (*obj*) [C Function]

Return `#t` iff *obj* is a thread; otherwise, return `#f`.

`join-thread` *thread* [*timeout* [*timeoutval*]] [Scheme Procedure]
`scm_join_thread` (*thread*) [C Function]
`scm_join_thread_timed` (*thread, timeout, timeoutval*) [C Function]

Wait for *thread* to terminate and return its exit value. Threads that have not been created with `call-with-new-thread` or `scm_spawn_thread` have an exit value of `#f`. 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* [Scheme Procedure]
`scm_thread_exited_p` (*thread*) [C Function]

Return `#t` iff *thread* has exited.

`yield` [Scheme Procedure]

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* [Scheme Procedure]
`scm_cancel_thread` (*thread*) [C Function]

Asynchronously notify *thread* to exit. Immediately after receiving this notification, *thread* will call its cleanup handler (if one has been set) and then terminate, aborting any evaluation that is in progress.

Because Guile threads are isomorphic with POSIX threads, *thread* will not receive its cancellation signal until it reaches a cancellation point. See your operating system's POSIX threading documentation for more information on cancellation points; note that in Guile, unlike native POSIX threads, a thread can receive a cancellation notification while attempting to lock a mutex.

`set-thread-cleanup!` *thread proc* [Scheme Procedure]
`scm_set_thread_cleanup_x` (*thread, proc*) [C Function]

Set *proc* as the cleanup handler for the thread *thread*. *proc*, which must be a thunk, will be called when *thread* exits, either normally or by being canceled. Thread cleanup handlers can be used to perform useful tasks like releasing resources, such as locked mutexes, when thread exit cannot be predicted.

The return value of *proc* will be set as the *exit value* of *thread*.

To remove a cleanup handler, pass `#f` for *proc*.

`thread-cleanup` *thread* [Scheme Procedure]
`scm_thread_cleanup` (*thread*) [C Function]

Return the cleanup handler currently installed for the thread *thread*. If no cleanup handler is currently installed, thread-cleanup returns `#f`.

Higher level thread procedures are available by loading the `(ice-9 threads)` module. These provide standardized thread creation.

`make-thread` *proc arg* ... [macro]

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* ... [macro]

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.

6.21.4 Mutexes and Condition Variables

A mutex is a thread synchronization object, it can be used by threads to control access to a shared resource. A mutex can be locked to indicate a resource is in use, and other threads can then block on the mutex to wait for the resource (or can just test and do something else if not available). "Mutex" is short for "mutual exclusion".

There are two types of mutexes in Guile, "standard" and "recursive". They're created by `make-mutex` and `make-recursive-mutex` respectively, the operation functions are then common to both.

Note that for both types of mutex there's no protection against a "deadly embrace". For instance if one thread has locked mutex A and is waiting on mutex B, but another thread

owns B and is waiting on A, then an endless wait will occur (in the current implementation). Acquiring requisite mutexes in a fixed order (like always A before B) in all threads is one way to avoid such problems.

`make-mutex` *flag* ... [Scheme Procedure]
`scm_make_mutex` () [C Function]
`scm_make_mutex_with_flags` (*SCM flags*) [C Function]

Return a new mutex. It is initially unlocked. If *flag* ... is specified, it must be a list of symbols specifying configuration flags for the newly-created mutex. The supported flags are:

`unchecked-unlock`
: Unless this flag is present, a call to 'unlock-mutex' on the returned mutex when it is already unlocked will cause an error to be signalled.

`allow-external-unlock`
: Allow the returned mutex to be unlocked by the calling thread even if it was originally locked by a different thread.

`recursive`
: The returned mutex will be recursive.

`mutex?` *obj* [Scheme Procedure]
`scm_mutex_p` (*obj*) [C Function]

Return `#t` iff *obj* is a mutex; otherwise, return `#f`.

`make-recursive-mutex` [Scheme Procedure]
`scm_make_recursive_mutex` () [C Function]

Create a new recursive mutex. It is initially unlocked. Calling this function is equivalent to calling 'make-mutex' and specifying the `recursive` flag.

`lock-mutex` *mutex* [*timeout* [*owner*]] [Scheme Procedure]
`scm_lock_mutex` (*mutex*) [C Function]
`scm_lock_mutex_timed` (*mutex, timeout, owner*) [C Function]

Lock *mutex*. 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.

When *owner* is given, it specifies an owner for *mutex* other than the calling thread. *owner* may also be `#f`, indicating that the mutex should be locked but left unowned.

For standard mutexes (`make-mutex`), and 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.

If *mutex* was locked by a thread that exited before unlocking it, the next attempt to lock *mutex* will succeed, but `abandoned-mutex-error` will be signalled.

When a system async (see Section 6.21.2.1 [System asyncs], page 397) is activated for a thread blocked in `lock-mutex`, the wait is interrupted and the async is executed. When the async returns, the wait resumes.

`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.

`try-mutex` *mx* [Scheme Procedure]
`scm_try_mutex` (*mx*) [C Function]
Try to lock *mutex* as per `lock-mutex`. If *mutex* can be acquired immediately then this is done and the return is `#t`. If *mutex* is locked by some other thread then nothing is done and the return is `#f`.

`unlock-mutex` *mutex* [*condvar* [*timeout*]] [Scheme Procedure]
`scm_unlock_mutex` (*mutex*) [C Function]
`scm_unlock_mutex_timed` (*mutex, condvar, timeout*) [C Function]
Unlock *mutex*. An error is signalled if *mutex* is not locked and was not created with the `unchecked-unlock` flag set, or if *mutex* is locked by a thread other than the calling thread and was not created with the `allow-external-unlock` flag set.

If *condvar* is given, it specifies a condition variable upon which the calling thread will wait to be signalled before returning. (This behavior is very similar to that of `wait-condition-variable`, except that the mutex is left in an unlocked state when the function returns.)

When *timeout* is also 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. Otherwise the function returns `#t`.

`mutex-owner` *mutex* [Scheme Procedure]
`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` *mutex* [Scheme Procedure]
`scm_mutex_level` (*mutex*) [C Function]
Return the current lock level of *mutex*. If *mutex* is currently unlocked, this value will be 0; otherwise, it will be the number of times *mutex* has been recursively locked by its current owner.

`mutex-locked?` *mutex* [Scheme Procedure]
`scm_mutex_locked_p` (*mutex*) [C Function]
Return `#t` if *mutex* is locked, regardless of ownership; otherwise, return `#f`.

`make-condition-variable` [Scheme Procedure]
`scm_make_condition_variable` () [C Function]
Return a new condition variable.

`condition-variable?` *obj* [Scheme Procedure]
`scm_condition_variable_p` (*obj*) [C Function]

Return `#t` iff *obj* is a condition variable; otherwise, return `#f`.

`wait-condition-variable` *condvar mutex* [*time*] [Scheme Procedure]
`scm_wait_condition_variable` (*condvar, mutex, time*) [C Function]

Wait until *condvar* has been signalled. While waiting, *mutex* is atomically unlocked (as with `unlock-mutex`) and is locked again when this function returns. When *time* is given, it specifies a point in time where the waiting should be aborted. It can be either a 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 a system 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` *condvar* [Scheme Procedure]
`scm_signal_condition_variable` (*condvar*) [C Function]

Wake up one thread that is waiting for *condvar*.

`broadcast-condition-variable` *condvar* [Scheme Procedure]
`scm_broadcast_condition_variable` (*condvar*) [C Function]

Wake up all threads that are waiting for *condvar*.

The following are higher level operations on mutexes. These are available from

```
(use-modules (ice-9 threads))
```

`with-mutex` *mutex body1 body2* ... [macro]

Lock *mutex*, evaluate the body *body1 body2* ..., then unlock *mutex*. The return value is that returned by the last body form.

The lock, body and unlock form the branches of a `dynamic-wind` (see Section 6.13.10 [Dynamic Wind], page 298), so *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.

6.21.5 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]

`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 a system async causes this function to be interrupted with error code `EINTR`.

`unsigned int scm_std_sleep` (*unsigned int seconds*) [C Function]

Like `sleep`, but leaves guile mode while sleeping. Also, the delivery of a system 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 a system async causes this function to be interrupted.

6.21.6 Critical Sections

`SCM_CRITICAL_SECTION_START` [C Macro]
`SCM_CRITICAL_SECTION_END` [C Macro]

These two macros can be used to delimit a critical section. Syntactically, they are both statements and need to be followed immediately by a semicolon.

Executing `SCM_CRITICAL_SECTION_START` will lock a recursive mutex and block the executing of system asyncs. Executing `SCM_CRITICAL_SECTION_END` will unblock the execution of system asyncs and unlock the mutex. Thus, the code that executes between these two macros can only be executed in one thread at any one time and no system asyncs will run. However, because the mutex is a recursive one, the code might still be reentered by the same thread. You must either allow for this or avoid it, both by careful coding.

On the other hand, critical sections delimited with these macros can be nested since the mutex is recursive.

You must make sure that for each `SCM_CRITICAL_SECTION_START`, the corresponding `SCM_CRITICAL_SECTION_END` is always executed. This means that no non-local exit (such as a signalled error) might happen, for example.

`void scm_dynwind_critical_section` (*SCM mutex*) [C Function]

Call `scm_dynwind_lock_mutex` on *mutex* and call `scm_dynwind_block_asyncs`. When *mutex* is false, a recursive mutex provided by Guile is used instead.

The effect of a call to `scm_dynwind_critical_section` is that the current dynwind context (see Section 6.13.10 [Dynamic Wind], page 298) turns into a critical section. Because of the locked mutex, no second thread can enter it concurrently and because of the blocked asyncs, no system async can reenter it from the current thread.

When the current thread reenters the critical section anyway, the kind of *mutex* determines what happens: When *mutex* is recursive, the reentry is allowed. When it is a normal mutex, an error is signalled.

6.21.7 Fluids and Dynamic States

A *fluid* is an object that can store one value per *dynamic state*. Each thread has a current dynamic state, and when accessing a fluid, this current dynamic state is used to provide the actual value. In this way, fluids can be used for thread local storage, but they are in fact more flexible: dynamic states are objects of their own and can be made current for more than one thread at the same time, or only be made current temporarily, for example.

Fluids can also be used to simulate the desirable effects of 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.

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!`.

`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` iff *obj* is a fluid; otherwise, return `#f`.

`fluid-ref` *fluid* [Scheme Procedure]
`scm_fluid_ref` (*fluid*) [C Function]

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* [Scheme Procedure]
`scm_fluid_set_x` (*fluid, value*) [C Function]

Set the value associated with *fluid* in the current dynamic root.

`fluid-unset!` *fluid* [Scheme Procedure]
`scm_fluid_unset_x` (*fluid*) [C Function]

Disassociate the given fluid from any value, making it unbound.

`fluid-bound?` *fluid* [Scheme Procedure]
`scm_fluid_bound_p` (*fluid*) [C Function]

Returns `#t` iff 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* [Scheme Procedure]
`scm_with_fluid` (*fluid, value, thunk*) [C Function]

Set *fluid* to *value* temporarily, and call *thunk*. *thunk* must be a procedure with no argument.

`with-fluids*` *fluids values thunk* [Scheme Procedure]
`scm_with_fluids` (*fluids, values, thunk*) [C Function]

Set *fluids* to *values* temporary, 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.

`SCM scm_c_with_fluids` (*SCM fluids, SCM vals, SCM* (**cproc*)(*void **), *void *data*) [C Function]

`SCM scm_c_with_fluid` (*SCM fluid, SCM val, SCM* (**cproc*)(*void **), *void *data*) [C Function]

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.

`void scm_dynwind_fluid` (*SCM fluid, SCM val*) [C Function]

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 298). 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.

`make-dynamic-state` [*parent*] [Scheme Procedure]

`scm_make_dynamic_state` (*parent*) [C Function]

Return a copy of the dynamic state object *parent* or of the current dynamic state when *parent* is omitted.

`dynamic-state?` *obj* [Scheme Procedure]

`scm_dynamic_state_p` (*obj*) [C Function]

Return `#t` if *obj* is a dynamic state object; return `#f` otherwise.

`int scm_is_dynamic_state` (*SCM obj*) [C Procedure]

Return non-zero if *obj* is a dynamic state object; return zero otherwise.

`current-dynamic-state` [Scheme Procedure]

`scm_current_dynamic_state` () [C Function]

Return the current dynamic state object.

`set-current-dynamic-state` *state* [Scheme Procedure]

`scm_set_current_dynamic_state` (*state*) [C Function]

Set the current dynamic state object to *state* and return the previous current dynamic state object.

`with-dynamic-state` *state proc* [Scheme Procedure]

`scm_with_dynamic_state` (*state, proc*) [C Function]

Call *proc* while *state* is the current dynamic state object.

`void scm_dynwind_current_dynamic_state` (*SCM state*) [C Procedure]

Set the current dynamic state to *state* for the current dynwind context.

`void * scm_c_with_dynamic_state` (*SCM state, void *(*func)(void *), void *data*) [C Procedure]

Like `scm_with_dynamic_state`, but call *func* with *data*.

6.21.8 Parameters

A parameter object is a procedure. Calling it with no arguments returns its value. Calling it with one argument sets the value.

```
(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 scope of the `parameterize` body. Leaving restores the previous locations. Re-entering (through a saved continuation) will again use the new locations.

```
(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. And introducing a new setting to existing code is often easier with a parameter object than adding arguments.

`make-parameter` *init* [*converter*] [Function]

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,

```
(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 *param*s bound to new locations and set to the given *value*s. *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 6.21.7 [Fluids and Dynamic States], page 405), 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.

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 7.5.26 [SRFI-39], page 583).

6.21.9 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)))
⇒ 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`.

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 6.21.4 [Mutexes and Condition Variables], page 400).

Guile's futures are implemented on top of POSIX threads (see Section 6.21.3 [Threads], page 398). 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 473).

`future` *exp* [Scheme Syntax]

Return a future for expression *exp*. This is equivalent to:

```
(make-future (lambda () exp))
```

`make-future` *thunk* [Scheme Procedure]

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* [Scheme Procedure]

Return `#t` if *obj* is a future.

`touch` *f* [Scheme Procedure]

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.

6.21.10 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 6.21.9 [Futures], page 409). 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* ... [syntax]

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 290).

`letpar` ((*var expr*) ...) *body1 body2* ... [syntax]

Evaluate each *expr* in parallel, each in its own thread, then bind the results to the corresponding *var* variables, and then evaluate *body1 body2* ...

`letpar` is like `let` (see Section 6.12.2 [Local Bindings], page 277), but all the expressions for the bindings are evaluated in parallel.

`par-map` *proc lst1 lst2* ... [Scheme Procedure]
`par-for-each` *proc lst1 lst2* ... [Scheme Procedure]

Call *proc* on the elements of the given lists. `par-map` returns a list comprising the return values from *proc*. `par-for-each` returns an unspecified value, but waits for all calls to complete.

The *proc* calls are `(proc elem1 elem2 ...)`, where each *elem* is from the corresponding *lst* . Each *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 `map` and `for-each` (see Section 6.7.2.8 [List Mapping], page 190), but make their *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 `current-processor-count` (see Section 7.2.7 [Processes], page 473). 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.

`n-par-map` *n proc lst1 lst2* ... [Scheme Procedure]
`n-par-for-each` *n proc lst1 lst2* ... [Scheme Procedure]

Call *proc* on the elements of the given lists, in the same way as `par-map` and `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 *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.

`n-for-each-par-map` *n sproc pproc lst1 lst2* ... [Scheme Procedure]

Apply *pproc* to the elements of the given lists, and apply *sproc* to each result returned by *pproc*. The final return value is unspecified, but all calls will have been completed before returning.

The calls made are `(sproc (pproc elem1 ... elemN))`, where each *elem* is from the corresponding *lst*. Each *lst* must have the same number of elements.

The *pproc* calls are made in parallel, in separate threads. No more than *n* threads are used at any one time. The order in which *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 `n-for-each-par-map` is like a combination of `n-par-map` and `for-each`,

```
(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.

6.22 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.

6.22.1 Configuration, Build and Installation

The following procedures and variables provide information about how Guile was configured, built and installed on your system.

`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]

Return a string describing Guile's full version number, effective version number, major, minor or micro version number, respectively. The `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/2.0/`'

```
(version) ⇒ "2.0.4"
(effective-version) ⇒ "2.0"
(major-version) ⇒ "2"
(minor-version) ⇒ "0"
(micro-version) ⇒ "4"
```

`%package-data-dir` [Scheme Procedure]
`scm_sys_package_data_dir ()` [C Function]

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 ()` [C Function]

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/2.0`'.

`%site-dir` [Scheme Procedure]
`scm_sys_site_dir ()` [C Function]

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`'.

`%guile-build-info` [Variable]

Alist 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, top_srcdir, prefix, exec_prefix, bindir, sbindir, libexecdir, datadir, sysconfdir, sharedstatedir, localstatedir, libdir, 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-2.0` 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*).

6.22.2 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.

6.22.2.1 Feature Manipulation

To check whether a particular feature is available, use the `provided?` procedure:

`provided?` *feature* [Scheme Procedure]
`feature?` *feature* [Deprecated Scheme Procedure]

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:

`provide` *feature* [Scheme Procedure]

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:

`void scm_add_feature` (*const char *str*) [C Function]

Add a symbol with name *str* to the list of available features in this Guile process.

6.22.2.2 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 6.7.6 [Arrays], page 198).

`array-for-each`
Indicates availability of `array-for-each` and other array mapping procedures (see Section 6.7.6 [Arrays], page 198).

`char-ready?`
Indicates that the `char-ready?` function is available (see Section 6.14.2 [Reading], page 306).

`complex` Indicates support for complex numbers.

`current-time`
Indicates availability of time-related functions: `times`, `get-internal-run-time` and so on (see Section 7.2.5 [Time], page 468).

`debug-extensions`
Indicates that the debugging evaluator is available, together with the options for controlling it.

`delay` Indicates support for promises (see Section 6.17.9 [Delayed Evaluation], page 356).

`EIDs` Indicates that the `geteuid` and `getegid` really return effective user and group IDs (see Section 7.2.7 [Processes], page 473).

`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 453).

`net-db` Indicates availability of network database functions: `scm_gethost`, `scm_getnet`, `scm_getproto`, `scm_getserv`, `scm_sethost`, `scm_setnet`, `scm_setproto`, `scm_setserv`, and their 'byXXX' variants (see Section 7.2.11.2 [Network Databases], page 485).

`posix` Indicates support for POSIX functions: `pipe`, `getgroups`, `kill`, `execl` and so on (see Section 7.2 [POSIX], page 452).

`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 336).

`socket` Indicates availability of socket-related functions: `socket`, `bind`, `connect` and so on (see Section 7.2.11.4 [Network Sockets and Communication], page 493).

`sort` Indicates availability of sorting and merging functions (see Section 6.11.3 [Sorting], page 268).

`system` Indicates that the `system` function is available (see Section 7.2.7 [Processes], page 473).

`threads` Indicates support for multithreading (see Section 6.21.3 [Threads], page 398).

`values` Indicates support for multiple return values using `values` and `call-with-values` (see Section 6.13.7 [Multiple Values], page 290).

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 248).

`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 628).

`record` Indicates support for record definition using `make-record-type` and friends (see Section 6.7.8 [Records], page 212).

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?`.

6.22.3 Runtime Options

There are a number of runtime options available for paramaterizing built-in procedures, like `read`, and built-in behavior, like what happens on an uncaught error.

For more information on reader options, See Section 6.17.2 [Scheme Read], page 346.

For more information on print options, See Section 6.17.3 [Scheme Write], page 347.

Finally, for more information on debugger options, See Section 6.25.3.4 [Debug Options], page 438.

6.22.3.1 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
<unknown-location>: warning: possibly unbound variable ‘aBc’
ERROR: In procedure module-lookup:
ERROR: Unbound variable: aBc
Entering a new prompt.  Type ‘,bt’ for a backtrace or ‘,q’ to continue.
scheme@(guile-user) [1]> (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.
scheme@(guile-user) [1]> (read-enable 'case-insensitive)
$2 = (square-brackets keywords #f case-insensitive positions)
scheme@(guile-user) [1]> ,q
scheme@(guile-user)> aBc
$3 = "hello"
```

6.23 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 10.4 [Compiling to the Virtual Machine], page 757.)

6.23.1 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 45). 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 6.17.5 [Compilation], page 350.

For more details on introspecting aspects of different languages, See Section 10.4.1 [Compiler Tower], page 758.

6.23.2 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.

6.23.2.1 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`.

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:

```
(boolean? #nil) ⇒ #t
(not #nil) ⇒ #t
(null? #nil) ⇒ #t
```

And in C, one has:

```
scm_is_bool (SCM_ELISP_NIL) ⇒ 1
scm_is_false (SCM_ELISP_NIL) ⇒ 1
scm_is_null (SCM_ELISP_NIL) ⇒ 1
```

In this way, a version of `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 `fold` written in Elisp can fold over a `'()`-terminated list, as made by Scheme.

On a low level, the bit representations for `#f`, `#t`, `nil`, and `'()` are made in such a way that they differ by only one bit, and so a test for, for example, `#f`-or-`nil` may be made very efficiently. See `libguile/boolean.h`, for more information.

6.23.2.2 Equality

Since Scheme's `equal?` must be transitive, and `'()` is not `equal?` to `#f`, to Scheme `nil` is not `equal?` to `#f` or `'()`.

```
(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
```

However, in Elisp, `'()`, `#f`, and `nil` are all `equal` (though not `eq`).

```
(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
```

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:

```
(define (truthiness x)
  (if (eq? x #f)
      #f
      #t))
```

This code seems to be meant to test a value for truth, but now that there are two false values, `#f` and `nil`, it is no longer correct.

Similarly, there is the loop:

```
(define (my-length l)
  (let lp ((l l) (len 0))
    (if (eq? l '())
        len
        (lp (cdr l) (1+ len)))))
```

Here, `my-length` will raise an error if *l* is a `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 *properties* of falsehood or nullity, not the individual members of that set. That is to say, they should use `not` or `null?` to test for falsehood or nullity, not `eq?` or `memv` or the like.

Fortunately, using `not` and `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:

```
(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)))))
```

This problem has a mirror-image case in Elisp:

```
(deffn my-falsep (x)
  (if (eq x nil)
      t
      nil))
```

Guile can warn when compiling code that has equality comparisons with `#f`, `'()`, or `nil`. See Section 6.17.5 [Compilation], page 350, for details.

6.23.2.3 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 6.21.7 [Fluids and Dynamic States], page 405, for more information.

6.23.2.4 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!

6.23.3 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`.

6.24 Support for Internationalization

Guile provides internationalization[15] 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.

[15] For concision and style, programmers often like to refer to internationalization as "i18n".

6.24.1 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 501 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 501).

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 501). 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 (i.e., the settings of the default `C` locale which usually represents conventions in use in the USA):

```
(make-locale (list LC_MESSAGE LC_CTYPE) "sv_SE")
```

The following example combines the use of Esperanto messages and conventions with monetary conventions from Croatia:

```
(make-locale LC_MONETARY "hr_HR"
             (make-locale LC_ALL "eo_EO"))
```

A `system-error` exception (see Section 6.13.11 [Handling Errors], page 301) is raised by `make-locale` when *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.

`locale?` *obj* [Scheme Procedure]
`scm_locale_p` (*obj*) [C Function]

Return true if *obj* is a locale object.

`%global-locale` [Scheme Variable]
`scm_global_locale` [C Variable]

This variable is bound to a locale object denoting the current process locale as installed using `setlocale ()` (see Section 7.2.13 [Locales], page 501). It may be used like any other locale object, including as a third argument to `make-locale`, for instance.

6.24.2 Text Collation

The following procedures provide support for text collation, i.e., locale-dependent string and character sorting.

`string-locale<?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_lt` (*s1, s2, locale*) [C Function]
`string-locale>?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_gt` (*s1, s2, locale*) [C Function]
`string-locale-ci<?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_lt` (*s1, s2, locale*) [C Function]
`string-locale-ci>?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_gt` (*s1, s2, locale*) [C Function]

Compare strings *s1* and *s2* in a locale-dependent way. If *locale* is provided, it should be locale object (as returned by `make-locale`) and will be used to perform the comparison; otherwise, the current system locale is used. For the `-ci` variants, the comparison is made in a case-insensitive way.

`string-locale-ci=?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_eq` (*s1, s2, locale*) [C Function]

Compare strings *s1* and *s2* in a case-insensitive, and locale-dependent way. If *locale* is provided, it should be a locale object (as returned by `make-locale`) and will be used to perform the comparison; otherwise, the current system locale is used.

`char-locale<?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_lt` (*c1, c2, locale*) [C Function]
`char-locale>?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_gt` (*c1, c2, locale*) [C Function]
`char-locale-ci<?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_lt` (*c1, c2, locale*) [C Function]
`char-locale-ci>?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_gt` (*c1, c2, locale*) [C Function]

Compare characters *c1* and *c2* according to either *locale* (a locale object as returned by `make-locale`) or the current locale. For the `-ci` variants, the comparison is made in a case-insensitive way.

`char-locale-ci=?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_eq` (*c1, c2, locale*) [C Function]

Return true if character *c1* is equal to *c2*, in a case insensitive way according to *locale* or to the current locale.

6.24.3 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 153). 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 "i" to "Latin capital letter I with dot above". The following procedures allow programmers to provide idiomatic character mapping.

`char-locale-downcase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_upcase` (*chr, locale*) [C Function]

Return the lowercase character that corresponds to *chr* according to either *locale* or the current locale.

`char-locale-upcase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_downcase` (*chr, locale*) [C Function]

Return the uppercase character that corresponds to *chr* according to either *locale* or the current locale.

`char-locale-titlecase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_titlecase` (*chr, locale*) [C Function]

Return the titlecase character that corresponds to *chr* according to either *locale* or the current locale.

`string-locale-upcase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_upcase` (*str, locale*) [C Function]

Return a new string that is the uppercase version of *str* according to either *locale* or the current locale.

`string-locale-downcase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_downcase` (*str, locale*) [C Function]

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*] [Scheme Procedure]
`scm_string_locale_titlecase` (*str, locale*) [C Function]

Return a new string that is the titlecase version of *str* according to either *locale* or the current locale.

6.24.4 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*]] [Scheme Procedure]
`scm_locale_string_to_integer` (*str, base, locale*) [C Function]

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 290): 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*] [Scheme Procedure]
`scm_locale_string_to_inexact` (*str*, *locale*) [C Function]

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 290): 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*]] [Scheme Procedure]

Convert *number* (an inexact) into a string according to the cultural conventions of either *locale* (a locale object) or the current locale. 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*] [Scheme Procedure]

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*).

6.24.5 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 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 468) 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` [Scheme Procedure]
`locale-negative-sign-position` [Scheme Procedure]

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 6.24.6 [Gettext Support], page 427).

Example uses of some of these functions are the implementation of the `number->locale-string` and `monetary-amount->locale-string` procedures (see Section 6.24.4 [Number Input and Output], page 423), as well as that the SRFI-19 date and time conversion to/from strings (see Section 7.5.16 [SRFI-19], page 568).

6.24.6 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 6.22.2 [Feature Tracking], page 414).

`gettext` *msg* [*domain* [*category*]] [Scheme Procedure]
`scm_gettext` (*msg, domain, category*) [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 501).

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*]] [Scheme Procedure]
`scm_ngettext` (*msg, msgplural, n, domain, category*) [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 501).

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\n"
                       "~a files processed\n" n)
             n))

(done 1) ⊣ 1 file processed
(done 3) ⊣ 3 files processed
```

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*).

`textdomain` [*domain*] [Scheme Procedure]
`scm_textdomain` (*domain*) [C Function]

Get or set the default gettext domain. When called with no parameter the current domain is returned. When called with a parameter, *domain* is set as the current domain, and that new value returned. For example,

```
(textdomain "myprog")
⇒ "myprog"
```

`bindtextdomain` *domain* [*directory*] [Scheme Procedure]
`scm_bindtextdomain` (*domain, directory*) [C Function]

Get or set the directory under which to find message files for *domain*. When called without a *directory* the current setting is returned. When called with a *directory*, *directory* is set for *domain* and that new setting returned. For example,

```
(bindtextdomain "myprog" "/my/tree/share/locale")
⇒ "/my/tree/share/locale"
```

When using Autoconf/Automake, an application should arrange for the configured `localedir` to get into the program (by substituting, or by generating a config file) and set that for its domain. This ensures the catalogue can be found even when installed in a non-standard location.

`bind-textdomain-codeset` *domain* [*encoding*] [Scheme Procedure]
`scm_bind_textdomain_codeset` (*domain, encoding*) [C Function]

Get or set the text encoding to be used by `gettext` for messages from *domain*. *encoding* is a string, the name of a coding system, for instance `"8859_1"`. (On a Unix/POSIX system the `iconv` program can list all available encodings.)

When called without an *encoding* the current setting is returned, or `#f` if none yet set. When called with an *encoding*, it is set for *domain* and that new setting returned. For example,

```
(bind-textdomain-codeset "myprog")
⇒ #f
(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.

6.25 Debugging Infrastructure

In order to understand Guile's debugging facilities, you first need to understand a little about how Guile represent 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.

6.25.1 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.

6.25.1.1 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.

```
(make-stack #t)
⇒
#<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, prompt tag and `#t` 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 `#t`, an integer, a prompt tag, or a procedure. `#t` means to cut away all frames up to but excluding the first user module frame. An integer means to cut away exactly that number of frames. A prompt tag means to cut away all frames that are inside a prompt with the given tag. A procedure means to cut away all frames up to but excluding the application frame whose procedure matches the specified one.

Each *outer_cut_i* can be an integer, a prompt tag, or a procedure. An integer means to cut away that number of frames. A prompt tag means to cut away all frames that are outside a prompt with the given tag. A procedure means to cut away frames down to but excluding the application frame whose procedure matches the specified one.

If the *outer_cut_i* of the last pair is missing, it is taken as 0.

`start-stack` *id exp* [Scheme Syntax]

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.

6.25.1.2 Stacks

`stack?` *obj* [Scheme Procedure]
`scm_stack_p` (*obj*) [C Function]

Return `#t` if *obj* is a calling stack.

`stack-id` *stack* [Scheme Procedure]
`scm_stack_id` (*stack*) [C Function]

Return the identifier given to *stack* by `start-stack`.

`stack-length` *stack* [Scheme Procedure]
`scm_stack_length` (*stack*) [C Function]

Return the length of *stack*.

`stack-ref` *stack index* [Scheme Procedure]
`scm_stack_ref` (*stack, index*) [C Function]

Return the *index*'th frame from *stack*.

`display-backtrace` *stack port* [*first* [*depth* [*highlights*]]] [Scheme Procedure]
`scm_display_backtrace_with_highlights` (*stack, port, first, depth, highlights*) [C Function]
`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.

6.25.1.3 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*) [C Function]

Return the previous frame of *frame*, or `#f` if *frame* is the first frame in its stack.

`frame-procedure` *frame* [Scheme Procedure]
`scm_frame_procedure` (*frame*) [C Function]

Return the procedure for *frame*, or `#f` if no procedure is associated with *frame*.

`frame-arguments` *frame* [Scheme Procedure]
`scm_frame_arguments` (*frame*) [C Function]

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 10.3.2 [VM Concepts], page 739, 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 10.3.3 [Stack Layout], page 740, for more information.

`frame-num-locals` *frame* [Scheme Procedure]
`frame-local-ref` *frame i* [Scheme Procedure]
`frame-local-set!` *frame i val* [Scheme Procedure]

Accessors for the temporary values corresponding to *frame*'s procedure application. The first local is the first argument given to the procedure. After the arguments, there are the local variables, and after that temporary values. See Section 10.3.3 [Stack Layout], page 740, for more information.

`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.

6.25.2 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 6.17.2 [Scheme Read], page 346). 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!` *obj alist* [Scheme Procedure]
`scm_set_source_properties_x` (*obj, alist*) [C Function]
Install the association list *alist* as the source property list for *obj*.

`set-source-property!` *obj key datum* [Scheme Procedure]
`scm_set_source_property_x` (*obj, key, datum*) [C Function]
Set the source property of object *obj*, which is specified by *key* to *datum*. Normally, the key will be a symbol.

`source-properties` *obj* [Scheme Procedure]
`scm_source_properties` (*obj*) [C Function]
Return the source property association list of *obj*.

`source-property` *obj key* [Scheme Procedure]
`scm_source_property` (*obj, key*) [C Function]
Return the property specified by *key* from *obj*'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 258, for more.

Guile also defines a couple of convenience macros built on `syntax-source`:

`current-source-location` [Scheme Syntax]
Expands to the source properties corresponding to the location of the `(current-source-location)` form.

`current-filename` [Scheme Syntax]
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 260), and want to preserve source information, the following helper function might be useful to you:

`cons-source` *xorig x y* [Scheme Procedure]
`scm_cons_source` (*xorig, x, y*) [C Function]
Create and return a new pair whose car and cdr are *x* and *y*. Any source properties associated with *xorig* are also associated with the new pair.

6.25.3 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.

6.25.3.1 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:

```
(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 294 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:

```
(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:

```
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.

6.25.3.2 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[16] 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"

[16] In effect, it is the default program which is run when no commands or script file are specified on the Guile command line.

handler of a `catch`. (For the full story of how these alternatives differ from each other, see Section 6.13.8 [Exceptions], page 291. 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 ()
           ;; 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;
  ...
  scm_c_catch (SCM_BOOL_T,
               my_body_proc, body_data,
               my_handler_proc, handler_data,
               my_preunwind_proc, &captured_stack);
  ...
  if (captured_stack != SCM_BOOL_F)
  {
    /* Display or process the captured stack. */
    ...
  }
  ...
}
```

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 6.25.1.2 [Stacks], page 431 and Section 6.25.1.3 [Frames], page 431.

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.

6.25.3.3 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:

```
(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 294, for more information. However, we did print out a context at the time of the error, using the built-in procedure, `backtrace`.

`backtrace` [*highlights*] [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 on-error='debug*] [*#:post-error post-error='catch*] [*#:pass-keys pass-keys='(quit)*] [*#:trap-handler trap-handler='debug*] [Scheme Procedure]

Call a thunk in a context in which errors are handled.

There are four 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 6.25.4 [Traps], page 440, for more information.

pass-keys A set of keys to ignore, as a list.

6.25.3.4 Debug options

The behavior of the `backtrace` procedure and of the default error handler can be parameterized via the debug options.

`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@(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 `keywords` option must be set using `debug-set!`.

`debug-enable` *option-name* [Scheme Procedure]
`debug-disable` *option-name* [Scheme Procedure]
`debug-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.

Stack overflow

Stack overflow errors are caused by a computation trying to use more stack space than has been enabled by the `stack` option. There are actually two kinds of stack that can overflow, the C stack and the Scheme stack.

Scheme stack overflows can occur if Scheme procedures recurse too far deeply. An example would be the following recursive loop:

```
scheme@(guile-user)> (let lp () (+ 1 (lp)))
<unnamed port>:8:17: In procedure vm-run:
<unnamed port>:8:17: VM: Stack overflow
```

The default stack size should allow for about 10000 frames or so, so one usually doesn't hit this level of recursion. Unfortunately there is no way currently to make a VM with a bigger stack. If you are in this unfortunate situation, please file a bug, and in the meantime, rewrite your code to be tail-recursive (see Section 3.3.2 [Tail Calls], page 24).

The other limit you might hit would be C stack overflows. 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:

```
scheme@(guile-user)> (use-modules (system vm vm))
scheme@(guile-user)> (debug-set! stack 10000)
scheme@(guile-user)> (let lp () (call-with-vm (the-vm) lp))
ERROR: In procedure call-with-vm:
ERROR: Stack overflow
```

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)
```

But of course it's better to have your code operate without so much resource consumption, avoiding loops through C trampolines.

6.25.4 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.

6.25.4.1 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 270) that can be fired at different times, which may be accessed with the following procedures.

All hooks are called with one argument, the frame in question. See Section 6.25.1.3 [Frames], page 431. 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:

```
(use-modules (system vm vm))
```

The result of calling `the-vm` is usually passed as the *vm* argument to all of these procedures.

`vm-next-hook` *vm* [Scheme Procedure]
The hook that will be fired before an instruction is retired (and executed).

`vm-push-continuation-hook` *vm* [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` *vm* [Scheme Procedure]
The hook that will be fired before returning from a frame.

This hook is a bit trickier than the rest, in that there is a particular interpretation of the values on the stack. Specifically, the top value on the stack is the number of values being returned, and the next *n* values are the actual values being returned, with the last value highest on the stack.

`vm-apply-hook` *vm* [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` *vm* [Scheme Procedure]
The hook that will be called after aborting to a prompt. See Section 6.13.5 [Prompts], page 285. The stack will be in the same state as for `vm-pop-continuation-hook`.

`vm-restore-continuation-hook` *vm* [Scheme Procedure]
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` *vm* [Scheme Procedure]
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!` *vm level* [Scheme Procedure]
Set the "trace level" of the VM.

See Section 10.3 [A Virtual Machine for Guile], page 738, for more information on Guile's virtual machine.

6.25.4.2 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.

6.25.4.3 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 6.25.4.1 [VM Hooks], page 440), 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 6.25.4.5 [Trap States], page 446, 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.

`#:closure?`
For traps that depend on the current frame's procedure, this argument specifies whether to trap on the only the specific procedure given, or on any closure that has the given procedure's code. Defaults to `#f`.

`#: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*] [*#:closure?*] [Scheme Procedure]
A trap that calls *handler* when *proc* is applied.

`trap-in-procedure` *proc enter-handler exit-handler* [*#:current-frame*] [*#:vm*] [*#:closure?*] [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.

`trap-instructions-in-procedure` *proc next-handler exit-handler* [*#:current-frame*] [*#:vm*] [*#:closure?*] [Scheme Procedure]
A trap that calls *next-handler* for every instruction executed in *proc*, and *exit-handler* when execution leaves *proc*.

`trap-at-procedure-ip-in-range` *proc range handler* [*#:current-frame*] [*#:vm*] [*#:closure?*] [Scheme Procedure]
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.

`trap-at-source-location` *file user-line handler* [#:current-frame] [#:vm] [Scheme Procedure]

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.

`trap-frame-finish` *frame return-handler abort-handler* [#:vm] [Scheme Procedure]

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.

`trap-in-dynamic-extent` *proc enter-handler return-handler abort-handler* [#:vm] [#:closure?] [Scheme Procedure]

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.

`trap-calls-in-dynamic-extent` *proc apply-handler return-handler* [#:current-frame] [#:vm] [#:closure?] [Scheme Procedure]

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*.

`trap-instructions-in-dynamic-extent` *proc next-handler* [#:current-frame] [#:vm] [#:closure?] [Scheme Procedure]

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 return-handler* [#:vm] [Scheme Procedure]

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.

6.25.4.4 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: |  |  2
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 6.25.4.3 [Low-Level Traps], page 442) 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 #:key* (*calls? #t*) (*instructions? #f*) (*width 80*) (*vm* (*the-vm*)) [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 49, for more information on tracing at the REPL.

6.25.4.5 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* [Scheme Procedure]
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 6.25.4.6 [High-Level Traps], page 447 are used in preference to this one.

`list-traps` [Scheme Procedure]
List the current set of traps, both enabled and disabled. Returns a list of integers.

`trap-name` *idx* [Scheme Procedure]
Returns the name associated with trap *idx*, or `#f` if there is no such trap.

`trap-enabled?` *idx* [Scheme Procedure]
Returns `#t` if trap *idx* is present and enabled, or `#f` otherwise.

`enable-trap!` *idx* [Scheme Procedure]
Enables trap *idx*.

`disable-trap!` *idx* [Scheme Procedure]
Disables trap *idx*.

`delete-trap!` *idx* [Scheme Procedure]
Removes trap *idx*, disabling it first, if necessary.

6.25.4.6 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* [Scheme Procedure]
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 6.25.4.1 [VM Hooks], page 440) 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* [Scheme Procedure]
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* [Scheme Procedure]
Install a trap that will fire when *proc* is called.

This is a breakpoint.

`add-trace-at-procedure-call!` *proc* [Scheme Procedure]
Install a trap that will print a tracing message when *proc* is called. See Section 6.25.4.4 [Tracing Traps], page 444, for more information.

This is a tracepoint.

`add-trap-at-source-location!` *file user-line* [Scheme Procedure]

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* [Scheme Procedure]

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?*] [Scheme Procedure]

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 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` *vm thunk* [Scheme Procedure]

Run *thunk*, a zero-argument procedure, using *vm*; instrument *vm* to collect code coverage data. Return code coverage data and the values returned by *thunk*.

`coverage-data?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a *coverage data* object as returned by `with-code-coverage`.

`coverage-data->lcov` *data port #:key modules* [Scheme Procedure]

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:

```
(use-modules (system vm coverage)
             (system vm vm))
```

```
(call-with-values (lambda ()
                    (with-code-coverage (the-vm)
                      (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 451). 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:

```
(use-modules (ice-9 slib))
```

`require` can then be used in the usual way (see Section "Require" in *The SLIB Manual*). For example,

```
(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:

   ```
   $ 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.:

   ```
   ln -s /usr/local/lib/slib /usr/local/share/guile/2.0/slib
   ```
3. Use Guile to create the catalog file, e.g.,:

   ```
   # guile
   guile> (use-modules (ice-9 slib))
   guile> (require 'new-catalog)
   guile> (quit)
   ```

   The catalog data should now be in '`/usr/local/share/guile/2.0/slibcat`'.

   If instead you get an error such as:

   ```
   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:

```
(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 661).

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 291.

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

[Error Reporting], page 297). 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* [Function]

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 scsh, Section 7.17 [The Scheme shell (scsh)], page 661.

File ports are implemented using low-level operating system I/O facilities, with optional buffering to improve efficiency; see Section 6.14.9.1 [File Ports], page 314.

Note that some procedures (e.g., `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 `peek-char` and `unread-char`.

The `force-output` and `drain-input` procedures can be used to clear the buffers.

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 *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 `fdopen` or `fdes->ports`.

`port-revealed` *port* [Scheme Procedure]
`scm_port_revealed` (*port*) [C Function]

Return the revealed count for *port*.

`set-port-revealed!` *port rcount* [Scheme Procedure]
`scm_set_port_revealed_x` (*port, rcount*) [C Function]

Sets the revealed count for a *port* to *rcount*. The return value is unspecified.

`fileno` *port* [Scheme Procedure]
`scm_fileno` (*port*) [C Function]

Return the integer file descriptor underlying *port*. Does not change its revealed count.

`port->fdes` *port* [Scheme Procedure]

Returns the integer file descriptor underlying *port*. As a side effect the revealed count of *port* is incremented.

`fdopen` *fdes modes* [Scheme Procedure]
`scm_fdopen` (*fdes, modes*) [C Function]

Return a new port based on the file descriptor *fdes*. Modes are given by the string *modes*. The revealed count of the port is initialized to zero. The *modes* string is the same as that accepted by `open-file` (see Section 6.14.9.1 [File Ports], page 314).

`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]

Decrements 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 473) 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` [Variable]
Open the file write-only.

`O_RDWR` [Variable]
Open the file read/write.

`O_APPEND` [Variable]
Append to the file instead of truncating.

`O_CREAT` [Variable]
Create the file if it does not already exist.

See Section "File Status Flags" in *The GNU C Library Reference Manual*, for additional flags.

`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 6.14.4 [Closing], page 309), 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.

`unread-char` *char* [*port*] [Scheme Procedure]
`scm_unread_char` (*char, port*) [C Function]
Place *char* in *port* so that it will be read by the next read operation on that port. If called multiple times, the unread characters will be read again in "last-in, first-out" order (i.e. a stack). If *port* is not supplied, the current input port is used.

`unread-string` *str port* [Scheme Procedure]
Place the string *str* in *port* so that its characters will be read in subsequent read operations. If called multiple times, the unread characters will be read again in last-in first-out order. If *port* is not supplied, the current-input-port is used.

`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`.

`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 `dup2` system call, if *newfd* (an integer) is supplied, otherwise a `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 `dup2` that any ports using *newfd* are moved to a different file descriptor and have their revealed counts set to zero.

`dup->fdes` *fd_or_port* [*fd*] [Scheme Procedure]
`scm_dup_to_fdes` (*fd_or_port*, *fd*) [C Function]

Return a new integer file descriptor referring to the open file designated by *fd_or_port*, which must be either an open file port or a file descriptor.

`dup->inport` *port/fd* [*newfd*] [Scheme Procedure]

Returns a new input port using the new file descriptor.

`dup->outport` *port/fd* [*newfd*] [Scheme Procedure]

Returns a new output port using the new file descriptor.

`dup` *port/fd* [*newfd*] [Scheme Procedure]

Returns a new port if *port/fd* is a port, with the same mode as the supplied port, otherwise returns an integer file descriptor.

`dup->port` *port/fd mode* [*newfd*] [Scheme Procedure]

Returns a new port using the new file descriptor. *mode* supplies a mode string for the port (see Section 6.14.9.1 [File Ports], page 314).

`duplicate-port` *port modes* [Scheme Procedure]

Returns a new port which is opened on a duplicate of the file descriptor underlying *port*, with mode string *modes* as for Section 6.14.9.1 [File Ports], page 314. 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 `(dup->port port modes)`.

`redirect-port` *old_port new_port* [Scheme Procedure]
`scm_redirect_port` (*old_port*, *new_port*) [C Function]

This procedure takes two ports and duplicates the underlying file descriptor from *old_port* into *new_port*. The current file descriptor in *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.

`dup2` *oldfd newfd* [Scheme Procedure]
`scm_dup2` (*oldfd, newfd*) [C Function]

A simple wrapper for the `dup2` system call. Copies the file descriptor *oldfd* to descriptor number *newfd*, replacing the previous meaning of *newfd*. Both *oldfd* and *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-mode` *port* [Scheme Procedure]

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-for-each` *proc* [Scheme Procedure]
`scm_port_for_each` (*SCM proc*) [C Function]
`scm_c_port_for_each` (*void* (**proc*)(*void *, SCM*), *void *data*) [C 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.

`setvbuf` *port mode* [*size*] [Scheme Procedure]
`scm_setvbuf` (*port, mode, size*) [C Function]

Set the buffering mode for *port*. *mode* can be:

`_IONBF` [Variable]

non-buffered

`_IOLBF` [Variable]

line buffered

`_IOFBF` [Variable]

block buffered, using a newly allocated buffer of *size* bytes. If *size* is omitted, a default size will be used.

`fcntl` *port/fd cmd* [*value*] [Scheme Procedure]
`scm_fcntl` (*object, cmd, value*) [C 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` [Variable]

Duplicate the file descriptor, the same as `dup->fdes` above does.

F_GETFD [Variable]
F_SETFD [Variable]

Get or set flags associated with the file descriptor. The only flag is the following,

FD_CLOEXEC [Variable]

"Close on exec", meaning the file descriptor will be closed on an `exec` call (a successful such call). For example to set that flag,

```
(fcntl port F_SETFD FD_CLOEXEC)
```

Or better, set it but leave any other possible future flags unchanged,

```
(fcntl port F_SETFD (logior FD_CLOEXEC
                            (fcntl port F_GETFD)))
```

F_GETFL [Variable]
F_SETFL [Variable]

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.

```
(fcntl sock F_SETFL (logior O_NONBLOCK
                            (fcntl sock F_GETFL)))
```

F_GETOWN [Variable]
F_SETOWN [Variable]

Get or set the process ID of a socket's owner, for `SIGIO` signals.

`flock` *file operation* [Scheme Procedure]
`scm_flock` (*file*, *operation*) [C Function]

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 [Variable]

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 124). If `flock` would block an `EWOULDBLOCK` error is thrown (see Section 7.2.1 [Conventions], page 452).

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. It also returns if interrupted by a signal.

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.

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` [Variable]
Test for execute permission.

`F_OK` [Variable]
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*.

`stat` *object* [Scheme Procedure]
`scm_stat` (*object*) [C Function]

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).

The object returned by `stat` can be passed as a single parameter to the following procedures, all of which return integers:

`stat:dev` *st* [Scheme Procedure]

The device number containing the file.

`stat:ino` *st* [Scheme Procedure]

The file serial number, which distinguishes this file from all other files on the same device.

`stat:mode` *st* [Scheme Procedure]

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* [Scheme Procedure]

The number of hard links to the file.

`stat:uid` *st* [Scheme Procedure]

The user ID of the file's owner.

`stat:gid` *st* [Scheme Procedure]

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* [Scheme Procedure]
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* [Scheme Procedure]
`scm_readlink` (*path*) [C Function]
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* [Scheme Procedure]
`scm_chown` (*object, owner, group*) [C Function]

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* [Scheme Procedure]
`scm_chmod` (*object, mode*) [C Function]

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*]]]]] [Scheme Procedure]
`scm_utime` (*pathname, actime, modtime, actimens, modtimens, flags*) [C Function]

`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.

`delete-file` *str* [Scheme Procedure]
`scm_delete_file` (*str*) [C Function]

Deletes (or "unlinks") the file whose path is specified by *str*.

`copy-file` *oldfile newfile* [Scheme Procedure]
`scm_copy_file` (*oldfile, newfile*) [C Function]

Copy the file specified by *oldfile* to *newfile*. The return value is unspecified.

`rename-file` *oldname newname* [Scheme Procedure]
`scm_rename` (*oldname, newname*) [C Function]

Renames the file specified by *oldname* to *newname*. The return value is unspecified.

`link` *oldpath newpath* [Scheme Procedure]
`scm_link` (*oldpath, newpath*) [C Function]

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* [Scheme Procedure]
`scm_symlink` (*oldpath, newpath*) [C Function]

Create a symbolic link named *newpath* with the value (i.e., pointing to) *oldpath*. The return value is unspecified.

`mkdir` *path* [*mode*] [Scheme Procedure]
`scm_mkdir` (*path, mode*) [C Function]

Create a new directory named by *path*. If *mode* is omitted then the permissions of the directory file are set using the current umask (see Section 7.2.7 [Processes], page 473). Otherwise they are set to the decimal value specified with *mode*. The return value is unspecified.

`rmdir` *path* [Scheme Procedure]
`scm_rmdir` (*path*) [C Function]

Remove the existing directory named by *path*. The directory must be empty for this to succeed. The return value is unspecified.

`opendir` *dirname* [Scheme Procedure]
`scm_opendir` (*dirname*) [C Function]

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 644).

`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 `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 `open` with `O_EXCL` to avoid any overwriting. A loop can try again with another name if the file exists (error `EEXIST`). `mkstemp!` below does that.

`mkstemp!` *tmpl* [Scheme Procedure]
`scm_mkstemp` (*tmpl*) [C Function]

Create a new unique file in the file system and return a new buffered port open for reading and writing to the file.

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. (`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 `chmod` to relax that if desired. For example `#o666` less `umask`, which is usual for ordinary file creation,

```
(let ((port (mkstemp! (string-copy "/tmp/myfile-XXXXXX"))))
  (chmod port (logand #o666 (lognot (umask))))
  ...)
```

`tmpfile` [Scheme Procedure]
`scm_tmpfile ()` [C Function]

Return an input/output port to a unique temporary file named using the path prefix `P_tmpdir` defined in '`stdio.h`'. The file is automatically deleted when the port is closed or the program terminates.

`dirname` *filename* [Scheme Procedure]
`scm_dirname` (*filename*) [C Function]

Return the directory name component of the file name *filename*. If *filename* does not contain a directory component, `.` is returned.

`basename` *filename* [*suffix*] [Scheme Procedure]
`scm_basename` (*filename, suffix*) [C Function]

Return the base name of the file name *filename*. The base name is the file name without any directory components. If *suffix* is provided, and is equal to the end of *basename*, it is removed also.

```
(basename "/tmp/test.xml" ".xml")
⇒ "test"
```

`file-exists?` *filename* [Scheme Procedure]

Return `#t` if the file named *filename* exists, `#f` if not.

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:

`passwd:name` *pw* [Scheme Procedure]

The name of the userid.

`passwd:passwd` *pw* [Scheme Procedure]

The encrypted passwd.

`passwd:uid` *pw* [Scheme Procedure]

The user id number.

`passwd:gid` *pw* [Scheme Procedure]

The group id number.

`passwd:gecos` *pw* [Scheme Procedure]

The full name.

`passwd:dir` *pw* [Scheme Procedure]

The home directory.

`passwd:shell` *pw* [Scheme Procedure]

The login shell.

`getpwuid` *uid* [Scheme Procedure]
Look up an integer userid in the user database.

`getpwnam` *name* [Scheme Procedure]
Look up a user name string in the user database.

`setpwent` [Scheme Procedure]
Initializes a stream used by `getpwent` to read from the user database. The next use of `getpwent` will return the first entry. The return value is unspecified.

`getpwent` [Scheme Procedure]
Read the next entry in the user database stream. The return is a passwd user object as above, or `#f` when no more entries.

`endpwent` [Scheme Procedure]
Closes the stream used by `getpwent`. The return value is unspecified.

`setpw` [*arg*] [Scheme Procedure]
`scm_setpwent` (*arg*) [C Function]
If called with a true argument, initialize or reset the password data stream. Otherwise, close the stream. The `setpwent` and `endpwent` procedures are implemented on top of this.

`getpw` [*user*] [Scheme Procedure]
`scm_getpwuid` (*user*) [C Function]
Look up an entry in the user database. *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:

`group:name` *gr* [Scheme Procedure]
The group name.

`group:passwd` *gr* [Scheme Procedure]
The encrypted group password.

`group:gid` *gr* [Scheme Procedure]
The group id number.

`group:mem` *gr* [Scheme Procedure]
A list of userids which have this group as a supplementary group.

`getgrgid` *gid* [Scheme Procedure]
Look up an integer group id in the group database.

`getgrnam` *name* [Scheme Procedure]
Look up a group name in the group database.

`setgrent` [Scheme Procedure]
Initializes a stream used by `getgrent` to read from the group database. The next use of `getgrent` will return the first entry. The return value is unspecified.

`getgrent` [Scheme Procedure]

Return the next entry in the group database, using the stream set by `setgrent`.

`endgrent` [Scheme Procedure]

Closes the stream used by `getgrent`. The return value is unspecified.

`setgr` [*arg*] [Scheme Procedure]
`scm_setgrent` (*arg*) [C Function]

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*] [Scheme Procedure]
`scm_getgrgid` (*group*) [C Function]

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` [Scheme Procedure]
`scm_getlogin` () [C Function]

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` [Scheme Procedure]
`scm_current_time` () [C Function]

Return the number of seconds since 1970-01-01 00:00:00 UTC, excluding leap seconds.

`gettimeofday` [Scheme Procedure]
`scm_gettimeofday` () [C Function]

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* [Scheme Procedure]
`set-tm:sec` *tm val* [Scheme Procedure]

Seconds (0-59).

`tm:min` *tm* [Scheme Procedure]
`set-tm:min` *tm val* [Scheme Procedure]

Minutes (0-59).

`tm:hour` *tm* [Scheme Procedure]
`set-tm:hour` *tm val* [Scheme Procedure]

Hours (0-23).

`tm:mday` *tm* [Scheme Procedure]
`set-tm:mday` *tm val* [Scheme Procedure]
Day of the month (1-31).

`tm:mon` *tm* [Scheme Procedure]
`set-tm:mon` *tm val* [Scheme Procedure]
Month (0-11).

`tm:year` *tm* [Scheme Procedure]
`set-tm:year` *tm val* [Scheme Procedure]
Year (70-), the year minus 1900.

`tm:wday` *tm* [Scheme Procedure]
`set-tm:wday` *tm val* [Scheme Procedure]
Day of the week (0-6) with Sunday represented as 0.

`tm:yday` *tm* [Scheme Procedure]
`set-tm:yday` *tm val* [Scheme Procedure]
Day of the year (0-364, 365 in leap years).

`tm:isdst` *tm* [Scheme Procedure]
`set-tm:isdst` *tm val* [Scheme Procedure]
Daylight saving indicator (0 for "no", greater than 0 for "yes", less than 0 for "unknown").

`tm:gmtoff` *tm* [Scheme Procedure]
`set-tm:gmtoff` *tm val* [Scheme Procedure]
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. $5 \times 60 \times 60$) in winter, or 14400 (ie. $4 \times 60 \times 60$) 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* [Scheme Procedure]
`set-tm:zone` *tm val* [Scheme Procedure]
Time zone label (a string), not necessarily unique.

`localtime` *time* [*zone*] [Scheme Procedure]
`scm_localtime` (*time*, *zone*) [C Function]
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* [Scheme Procedure]
`scm_gmtime` (*time*) [C Function]
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.

```
(strftime "%c" (localtime (current-time)))
⇒ "Mon Mar 11 20:17:43 2002"
```

If `setlocale` has been called (see Section 7.2.13 [Locales], page 501), 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*] [Scheme Procedure]
`scm_environ` (*env*) [C Function]

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* [Scheme Procedure]
`scm_putenv` (*str*) [C Function]

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* [Scheme Procedure]
`scm_chdir` (*str*) [C Function]

Change the current working directory to *str*. The return value is unspecified.

`getcwd` [Scheme Procedure]
`scm_getcwd` () [C Function]

Return the name of the current working directory.

`umask` [*mode*] [Scheme Procedure]
`scm_umask` (*mode*) [C Function]

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* [Scheme Procedure]
`scm_chroot` (*path*) [C Function]

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` [Scheme Procedure]
`scm_getpid` () [C Function]

Return an integer representing the current process ID.

`getgroups` [Scheme Procedure]
`scm_getgroups` () [C Function]

Return a vector of integers representing the current supplementary group IDs.

`getppid` [Scheme Procedure]
`scm_getppid ()` [C Function]
Return an integer representing the process ID of the parent process.

`getuid` [Scheme Procedure]
`scm_getuid ()` [C Function]
Return an integer representing the current real user ID.

`getgid` [Scheme Procedure]
`scm_getgid ()` [C Function]
Return an integer representing the current real group ID.

`geteuid` [Scheme Procedure]
`scm_geteuid ()` [C Function]
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` [Scheme Procedure]
`scm_getegid ()` [C Function]
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* [Scheme Procedure]
`scm_setgroups` (*vec*) [C Function]
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 Groups" in *The GNU C Library Reference Manual*).

`setuid` *id* [Scheme Procedure]
`scm_setuid` (*id*) [C Function]
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* [Scheme Procedure]
`scm_setgid` (*id*) [C Function]
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* [Scheme Procedure]
`scm_seteuid` (*id*) [C Function]
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 unspecified.

`setegid` *id* [Scheme Procedure]
`scm_setegid` (*id*) [C Function]

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` [Scheme Procedure]
`scm_getpgrp` () [C Function]

Return an integer representing the current process group ID. This is the POSIX definition, not BSD.

`setpgid` *pid pgid* [Scheme Procedure]
`scm_setpgid` (*pid, pgid*) [C Function]

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` [Scheme Procedure]
`scm_setsid` () [C Function]

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* [Scheme Procedure]
`scm_getsid` (*pid*) [C Function]

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*] [Scheme Procedure]
`scm_waitpid` (*pid, options*) [C Function]

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 *options* argument, if supplied, should be the bitwise OR of the values of zero or more of the following variables:

`WNOHANG` [Variable]
Return immediately even if there are no child processes to be collected.

`WUNTRACED` [Variable]
Report status information for stopped processes as well as terminated processes.

The return value is a pair containing:

1. The process ID of the child process, or 0 if `WNOHANG` was specified and no process was collected.
2. The integer status value.

The following three functions can be used to decode the process status code returned by `waitpid`.

`status:exit-val` *status* [Scheme Procedure]
`scm_status_exit_val` (*status*) [C Function]
Return the exit status value, as would be set if a process ended normally through a call to `exit` or `_exit`, if any, otherwise `#f`.

`status:term-sig` *status* [Scheme Procedure]
`scm_status_term_sig` (*status*) [C Function]
Return the signal number which terminated the process, if any, otherwise `#f`.

`status:stop-sig` *status* [Scheme Procedure]
`scm_status_stop_sig` (*status*) [C Function]
Return the signal number which stopped the process, if any, otherwise `#f`.

`system` [*cmd*] [Scheme Procedure]
`scm_system` (*cmd*) [C Function]
Execute *cmd* using the operating system's "command processor". Under Unix this is usually the default shell `sh`. The value returned is *cmd*'s exit status as returned by `waitpid`, which can be interpreted using the functions above.

If `system` is called without arguments, return a boolean indicating whether the command processor is available.

`system*` *arg1 arg2 ...* [Scheme Procedure]
`scm_system_star` (*args*) [C Function]
Execute the command indicated by *arg1 arg2 ...*. 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.

This function returns the exit status of the command as provided by `waitpid`. This value can be handled with `status:exit-val` and the related functions.

`system*` is similar to `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 `system` in situations where shell interpretation is not required.

Example: (system* `"echo" "foo" "bar"`)

`primitive-exit` [*status*] [Scheme Procedure]
`primitive-_exit` [*status*] [Scheme Procedure]
`scm_primitive_exit` (*status*) [C Function]
`scm_primitive__exit` (*status*) [C Function]

Terminate the current process without unwinding the Scheme stack. The exit status is *status* if supplied, otherwise zero.

`primitive-exit` uses the C `exit` function and hence runs usual C level cleanups (flush output streams, call `atexit` functions, etc, see Section "Normal Termination" in *The GNU C Library Reference Manual*)).

`primitive-_exit` is the `_exit` system call (see Section "Termination Internals" in *The GNU C Library Reference Manual*). This terminates the program immediately, with neither Scheme-level nor C-level cleanups.

The typical use for `primitive-_exit` is from a child process created with `primitive-fork`. For example in a Gdk program the child process inherits the X server connection and a C-level `atexit` cleanup which will close that connection. But closing in the child would upset the protocol in the parent, so `primitive-_exit` should be used to exit without that.

`execl` *filename arg* ... [Scheme Procedure]
`scm_execl` (*filename, args*) [C Function]

Executes the file named by *filename* as a new process image. The remaining arguments are supplied to the process; from a C program they are accessible as the `argv` argument to `main`. Conventionally the first *arg* is the same as *filename*. All arguments must be strings.

If *arg* is missing, *filename* is executed with a null argument list, which may have system-dependent side-effects.

This procedure is currently implemented using the `execv` system call, but we call it `execl` because of its Scheme calling interface.

`execlp` *filename arg* ... [Scheme Procedure]
`scm_execlp` (*filename, args*) [C Function]

Similar to `execl`, however if *filename* does not contain a slash then the file to execute will be located by searching the directories listed in the `PATH` environment variable.

This procedure is currently implemented using the `execvp` system call, but we call it `execlp` because of its Scheme calling interface.

`execle` *filename env arg* ... [Scheme Procedure]
`scm_execle` (*filename, env, args*) [C Function]

Similar to `execl`, 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` [Scheme Procedure]
`scm_fork` () [C Function]

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.

This procedure has been renamed from `fork` to avoid a naming conflict with the scsh fork.

`nice` *incr* [Scheme Procedure]
`scm_nice` (*incr*) [C Function]

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* [Scheme Procedure]
`scm_setpriority` (*which, who, prio*) [C Function]

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* [Scheme Procedure]
`scm_getpriority` (*which, who*) [C Function]

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* [Scheme Procedure]
`scm_getaffinity` (*pid*) [C Function]

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* [Scheme Procedure]
`scm_setaffinity` (*pid, mask*) [C Function]

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*).

`total-processor-count` [Scheme Procedure]
`scm_total_processor_count` () [C Function]

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` [Scheme Procedure]
`scm_current_processor_count ()` [C Function]

Like `total-processor-count`, but return the number of processors available to the current process. See `setaffinity` and `getaffinity` for more information.

7.2.8 Signals

The following procedures raise, handle and wait for signals.

Scheme code signal handlers are run via a system async (see Section 6.21.2.1 [System asyncs], page 397), 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* [Scheme Procedure]
`scm_kill` (*pid, sig*) [C Function]

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` [Variable]

Hang-up signal.

`SIGINT` [Variable]

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* [Scheme Procedure]
`scm_raise` (*sig*) [C Function]

Sends a specified signal *sig* to the current process, where *sig* is as described for the `kill` procedure.

`sigaction` *signum* [*handler* [*flags* [*thread*]]] [Scheme Procedure]
`scm_sigaction` (*signum, handler, flags*) [C Function]
`scm_sigaction_for_thread` (*signum, handler, flags, thread*) [C Function]

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 hander, 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 124) of the following (where provided by the system), or `0` for none.

`SA_NOCLDSTOP` [Variable]

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` [Variable]

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).

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]
`usleep` *usecs* [Scheme Procedure]
`scm_sleep` (*secs*) [C Function]
`scm_usleep` (*usecs*) [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 6.21.5 [Blocking], page 404).

`getitimer` *which_timer* [Scheme Procedure]
`setitimer` *which_timer interval_seconds interval_microseconds periodic_seconds periodic_microseconds* [Scheme Procedure]
`scm_getitimer` (*which_timer*) [C Function]
`scm_setitimer` (*which_timer, interval_seconds, interval_microseconds, periodic_seconds, periodic_microseconds*) [C Function]

Get or set the periods programmed in certain system timers. These timers have a current interval value which counts down and on reaching zero raises a signal. An optional periodic value can be set to restart from there each time, for periodic operation. *which_timer* is one of the following values

`ITIMER_REAL` [Variable]

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` [Variable]

A virtual-time timer, counting down while the current process is actually using CPU. At zero it raises `SIGVTALRM`.

`ITIMER_PROF` [Variable]

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 current timer value and its programmed restart value, as a list containing two pairs. Each pair is a time in seconds and microseconds: `((`*interval_secs* `.` *interval_usecs*`) (`*periodic_secs* `.` *periodic_usecs*`))`.

`setitimer` sets the timer values similarly, in seconds and microseconds (which must be integers). The periodic 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     ;; first SIGALRM in 5.5 seconds time
           2 0)         ;; then repeat every 2 seconds
```

Although the timers are programmed in microseconds, the actual accuracy might not be that high.

7.2.9 Terminals and Ptys

`isatty?` *port* [Scheme Procedure]
`scm_isatty_p` (*port*) [C Function]

Return `#t` if *port* is using a serial non–file device, otherwise `#f`.

`ttyname` *port* [Scheme Procedure]
`scm_ttyname` (*port*) [C Function]

Return a string with the name of the serial terminal device underlying *port*.

`ctermid` [Scheme Procedure]
`scm_ctermid` () [C Function]

Return a string containing the file name of the controlling terminal for the current process.

`tcgetpgrp` *port* [Scheme Procedure]
`scm_tcgetpgrp` (*port*) [C Function]

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* [Scheme Procedure]
`scm_tcsetpgrp` (*port, pgid*) [C Function]

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* [Scheme Procedure]
`open-pipe*` *mode prog* [*args...*] [Scheme Procedure]

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, ie. to read from the subprocess. `OPEN_WRITE` is an output pipe, ie. to write to it.

`OPEN_READ` [Variable]
`OPEN_WRITE` [Variable]
`OPEN_BOTH` [Variable]

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 cases the child's standard error is inherited from `current-error-port` (see Section 6.14.8 [Default Ports], page 313).

If those `current-X-ports` are not files of some kind, and hence don't have file descriptors for the child, then '`/dev/null`' is used instead.

Care should be taken with `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 `PIPE_BUF` bytes of buffering (see Section 7.2.2 [Ports and File Descriptors], page 453), which will be enough for small writes, but not for say putting a big file through a filter.

`open-input-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_READ`.

```
(let* ((port (open-input-pipe "date --utc"))
       (str  (read-line port)))
  (close-pipe port)
  str)
⇒ "Mon Mar 11 20:10:44 UTC 2002"
```

`open-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_WRITE`.

```
(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")))
```

`open-input-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_BOTH`.

`close-pipe` *port* [Scheme Procedure]

Close a pipe created by `open-pipe`, wait for the process to terminate, and return the wait status code. The status is as per `waitpid` and can be decoded with `status:exit-val` etc (see Section 7.2.7 [Processes], page 473)

`waitpid` `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 `close-pipe`.

`close-port` (see Section 6.14.4 [Closing], page 309) 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 `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.

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 493) 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, ie. '`127.0.0.1`'.

`inet-aton` *address* [Scheme Procedure]
`scm_inet_aton` (*address*) [C Function]

This function is deprecated in favor of `inet-pton`.

Convert an IPv4 Internet address from printable string (dotted decimal notation) to an integer. E.g.,

```
(inet-aton "127.0.0.1") ⇒ 2130706433
```

`inet-ntoa` *inetid* [Scheme Procedure]
`scm_inet_ntoa` (*inetid*) [C Function]

This function is deprecated in favor of `inet-ntop`.

Convert an IPv4 Internet address to a printable (dotted decimal notation) string. E.g.,

```
(inet-ntoa 2130706433) ⇒ "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.,

```
(inet-netof 2130706433) ⇒ 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.,

```
(inet-lnaof 2130706433) ⇒ 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.,

```
(inet-makeaddr 127 1) ⇒ 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.,

```
(inet-ntop AF_INET 2130706433) ⇒ "127.0.0.1"
(inet-ntop AF_INET6 (- (expt 2 128) 1))
  ⇒ "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
```

`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.,

```
(inet-pton AF_INET "127.0.0.1") ⇒ 2130706433
(inet-pton AF_INET6 "::1") ⇒ 1
```

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.

```
(let* ((ai (car (getaddrinfo "www.gnu.org" "http")))
       (s  (socket (addrinfo:fam ai) (addrinfo:socktype ai)
                   (addrinfo:protocol ai))))
  (connect s (addrinfo:addr ai))
  s)
```

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\n"
          (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; the error code can be found in `errno`.

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* [Scheme Procedure]
: Return flags for *ai* as a bitwise or of `AI_` values (see above).

`addrinfo:fam` *ai* [Scheme Procedure]
: Return the address family of *ai* (a `AF_` value).

`addrinfo:socktype` *ai* [Scheme Procedure]
: Return the socket type for *ai* (a `SOCK_` value).

`addrinfo:protocol` *ai* [Scheme Procedure]
: Return the protocol of *ai*.

`addrinfo:addr` *ai* [Scheme Procedure]

Return the socket address associated with *ai* as a `sockaddr` object (see Section 7.2.11.3 [Network Socket Address], page 492).

`addrinfo:canonname` *ai* [Scheme Procedure]

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* [Scheme Procedure]

The "official" hostname for *host*.

`hostent:aliases` *host* [Scheme Procedure]

A list of aliases for *host*.

`hostent:addrtype` *host* [Scheme Procedure]

The host address type, one of the `AF` constants, such as `AF_INET` or `AF_INET6`.

`hostent:length` *host* [Scheme Procedure]

The length of each address for *host*, in bytes.

`hostent:addr-list` *host* [Scheme Procedure]

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 484).

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]

Initialize an internal stream from which host objects may be read. This procedure must be called before any calls to `gethostent`, and may also be called afterward to reset the host entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `gethostbyname` or `gethostbyaddr` calls, possibly giving an efficiency gain.

`gethostent` [Scheme Procedure]

Return the next host object from the host database, or `#f` if there are no more hosts to be found (or an error has been encountered). This procedure may not be used before `sethostent` has been called.

`endhostent` [Scheme Procedure]

Close the stream used by `gethostent`. The return value is unspecified.

`sethost` [*stayopen*] [Scheme Procedure]
`scm_sethost` (*stayopen*) [C Function]

If *stayopen* is omitted, this is equivalent to `endhostent`. Otherwise it is equivalent to `sethostent stayopen`.

The Network Database

The following functions accept an object representing a network and return a selected component:

`netent:name` *net* [Scheme Procedure]

The "official" network name.

`netent:aliases` *net* [Scheme Procedure]

A list of aliases for the network.

`netent:addrtype` *net* [Scheme Procedure]

The type of the network number. Currently, this returns only `AF_INET`.

`netent:net` *net* [Scheme Procedure]

The network number.

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]
`getprotobyname` *name* [Scheme Procedure]
`getprotobynumber` *number* [Scheme Procedure]
`scm_getproto` (*protocol*) [C Function]
Look up a network protocol by name or by number. `getprotobyname` 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` [Scheme Procedure]
Return the next entry from the protocol database.

`endprotoent` [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 component:

`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* [Scheme Procedure]
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*]] [Scheme Procedure]
`getservbyname` *name protocol* [Scheme Procedure]
`getservbyport` *port protocol* [Scheme Procedure]
`scm_getserv` (*name, protocol*) [C Function]
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") 143 "tcp")

(getservbyport 88 "udp")
⇒ #("kerberos" ("kerberos5" "krb5") 88 "udp")
```

The following procedures may be used to step through the service database from beginning to end.

`setservent` [*stayopen*] [Scheme Procedure]

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` [Scheme Procedure]

Return the next entry from the services database.

`endservent` [Scheme Procedure]

Close the stream used by `getservent`. The return value is unspecified.

`setserv` [*stayopen*] [Scheme Procedure]
`scm_setserv` (*stayopen*) [C Function]

If *stayopen* is omitted, this is equivalent to `endservent`. Otherwise it is equivalent to `setservent stayopen`.

7.2.11.3 Network Socket Address

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 484), 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 484).

`struct sockaddr * scm_c_make_socket_address` (*SCM family, SCM address, SCM args, size_t *outsize*) [C Function]

Return a newly-`malloc`ed `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-`malloc`ed `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 7.2.2 [Ports and File Descriptors], page 453.

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 304).

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 485). A value of zero means the default protocol, which is usually right.

A socket cannot by 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 485).

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 *optname*s 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]

The *value* taken or returned is an integer.

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 *optname*s 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 484). *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.

`shutdown` *sock how* [Scheme Procedure]
`scm_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 492). The return value is unspecified.

```
(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 492). 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* [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, wait until one is available (unless `O_NONBLOCK` has been set on the socket, see Section 7.2.2 [Ports and File Descriptors], page 453).

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 492) which is where the remote connection is from (like `getpeername` below).

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* [Scheme Procedure]
`scm_getsockname` (*sock*) [C Function]

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* [Scheme Procedure]
`scm_getpeername` (*sock*) [C Function]

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*] [Scheme Procedure]
`scm_recv` (*sock, buf, flags*) [C Function]

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*] [Scheme Procedure]
`scm_send` (*sock, message, flags*) [C Function]

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*]]] [Scheme Procedure]
`scm_recvfrom` (*sock, buf, flags, start, end*) [C Function]

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 492) 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*] [Scheme Procedure]
`sendto` *sock message AF_INET ipv4addr port* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET6 ipv6addr port* [*flowinfo* [*scopeid* [*flags*]]] [Scheme Procedure]
`sendto` *sock message AF_UNIX path* [*flags*] [Scheme Procedure]
`scm_sendto` (*sock, message, fam, address, args_and_flags*) [C Function]

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 492).

The destination address may be followed by an optional *flags* argument which is a `logior` (see Section 6.6.2.13 [Bitwise Operations], page 124) 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.

The following functions can be used to convert short and long integers between "host" and "network" order. Although the procedures above do this automatically for addresses, the conversion will still need to be done when sending or receiving encoded integer data from the network.

`htons` *value* [Scheme Procedure]
`scm_htons` (*value*) [C Function]

Convert a 16 bit quantity from host to network byte ordering. *value* is packed into 2 bytes, which are then converted and returned as a new integer.

`ntohs` *value* [Scheme Procedure]
`scm_ntohs` (*value*) [C Function]

Convert a 16 bit quantity from network to host byte ordering. *value* is packed into 2 bytes, which are then converted and returned as a new integer.

`htonl` *value* [Scheme Procedure]
`scm_htonl` (*value*) [C Function]

Convert a 32 bit quantity from host to network byte ordering. *value* is packed into 4 bytes, which are then converted and returned as a new integer.

`ntohl` *value* [Scheme Procedure]
`scm_ntohl` (*value*) [C Function]

Convert a 32 bit quantity from network to host byte ordering. *value* is packed into 4 bytes, which are then converted and returned as a new integer.

These procedures are inconvenient to use at present, but consider:

```
(define write-network-long
  (lambda (value port)
    (let ((v (make-uniform-vector 1 1 0)))
      (uniform-vector-set! v 0 (htonl value))
      (uniform-vector-write v port))))

(define read-network-long
  (lambda (port)
    (let ((v (make-uniform-vector 1 1 0)))
      (uniform-vector-read! v port)
      (ntohl (uniform-vector-ref v 0)))))
```

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.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (connect s AF_INET (inet-pton AF_INET "127.0.0.1") 80)
  (display "GET / HTTP/1.0\r\n\r\n" s)

  (do ((line (read-line s) (read-line s)))
      ((eof-object? line))
    (display line)
    (newline)))
```

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* [Scheme Procedure]

The name of the operating system.

`utsname:nodename` *un* [Scheme Procedure]

The network name of the computer.

`utsname:release` *un* [Scheme Procedure]

The current release level of the operating system implementation.

`utsname:version` *un* [Scheme Procedure]
The current version level within the release of the operating system.

`utsname:machine` *un* [Scheme Procedure]
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 6.24.1 [i18n Introduction], page 421, 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]
`scm_crypt` (*key, salt*) [C Function]
Encrypt *key*, with the addition of *salt* (both strings), using the `crypt` C library call.

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]
`scm_getpass` (*prompt*) [C Function]

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[1] 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 525, 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.

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 568), 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 517), with each

[1] Yes, the P is for protocol, but this phrase appears repeatedly in RFC 2616.

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 8.4 [sxml simple], page 669, 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`:

```
(define (para . contents)
  `(p ,@contents))

(use-modules (sxml simple))
(sxml->xml (you-said "Hi!"))
⊣ <p>You said: Hi!</p>

(sxml->xml (you-said "<i>Rats, foiled again!</i>"))
⊣ <p>You said: &lt;i&gt;Rats, foiled again!&lt;/i&gt;</p>
```

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.

```
(sxml->xml (you-said (you-said "<Hi!>")))
⊣ <p>You said: <p>You said: &lt;Hi!&gt;</p></p>
```

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 := 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. All URIs have a scheme and a path (though the path might be empty). Some URIs have a host, and some of those have ports and userinfo. Any URI might have a query part or a 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*.

Properly speaking, a fragment is not part of a URI. For example, when a web browser follows a link to `http://example.com/#foo`, it sends a request for `http://example.com/`, then looks in the resulting page for the fragment identified `foo` reference. A fragment identifies a part of a resource, not the resource itself. But it is useful to have a fragment field in the URI record itself, so we hope you will forgive the inconsistency.

```
(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.

`build-uri` *scheme* [*#:userinfo=*`#f`] [*#:host=*`#f`] [*#:port=*`#f`] [*#:path=*`""`] [*#:query=*`#f`] [*#:fragment=*`#f`] [*#:validate?=*`#t`] [Scheme Procedure]

Construct a URI object. *scheme* should be a symbol, 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.

`uri?` *x* [Scheme Procedure]
`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]

A predicate and field accessors for the URI record type. The URI scheme will be a symbol, and the rest either strings or `#f` if not present.

`string->uri` *string* [Scheme Procedure]

Parse *string* into a URI object. Return `#f` if the string could not be parsed.

`uri->string` *uri* [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.

`declare-default-port!` *scheme port* [Scheme Procedure]

Declare a default port for the given URI scheme.

`uri-decode` *str* [*#:encoding=*`"utf-8"`] [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 304, for more information on character encodings.

Returns a string of the decoded characters, or a bytevector if *encoding* was `#f`.

Fixme: clarify return type. indicate default values. type of unescaped-chars.

`uri-encode` *str* [*#:encoding=*`"utf-8"`] [*#:unescaped-chars*] [Scheme Procedure]

Percent-encode any character not in the character set, *unescaped-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"`.

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 521, and see Section 7.3.9 [Web Server], page 522. Otherwise, load the HTTP module, and read on.

```
(use-modules (web http))
```

The focus of the `(web http)` module is to parse and unparse 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 568), 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* [Scheme Procedure]
Return `#t` iff *sym* is a known header, with associated parsers and serialization procedures.

`header-parser` *sym* [Scheme Procedure]
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* [Scheme Procedure]
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* [Scheme Procedure]
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 509. To add your own, use the `declare-header!` procedure:

`declare-header!` *name parser validator writer* [Scheme Procedure]
[*#: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-aton str))
  (lambda (ip)
    (and (integer? ip) (exact? ip) (<= 0 ip #xffffffff)))
  (lambda (ip port)
    (display (inet-ntoa ip) port)))
```

`declare-opaque-header!` *name* [Scheme Procedure]
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* [Scheme Procedure]
Return a true value iff *val* is a valid Scheme value for the header with name *sym*.

Now that we have a generic interface for reading and writing headers, we do just that.

`read-header` *port* [Scheme Procedure]

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* [Scheme Procedure]

Parse *val*, a string, with the parser for the header named *name*. Returns the parsed value.

`write-header` *name val port* [Scheme Procedure]

Write the given header name and value to *port*, using the writer from `header-writer`.

`read-headers` *port* [Scheme Procedure]

Read the headers of an HTTP message from *port*, returning the headers as an ordered alist.

`write-headers` *headers port* [Scheme Procedure]

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*] [Scheme Procedure]

Parse an HTTP method from *str*. The result is an upper-case symbol, like `GET`.

`parse-http-version` *str* [*start*] [*end*] [Scheme Procedure]

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*] [Scheme Procedure]

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* [Scheme Procedure]

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* [Scheme Procedure]

Write the first line of an HTTP request to *port*.

`read-response-line` *port* [Scheme Procedure]

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* [Scheme Procedure]

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` [HTTP Header Type]

A SRFI-19 date.

`KVList` [HTTP Header Type]

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` [HTTP Header Type]

A list of strings.

`Quality` [HTTP Header Type]

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` [HTTP Header Type]

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` [HTTP Header Type]

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` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

The date that a given HTTP message was originated.

```
(parse-header 'date "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`KVList pragma` [HTTP Header]

A key-value list of implementation-specific directives.

```
(parse-header 'pragma "no-cache, broccoli=tasty")
⇒ (no-cache (broccoli . "tasty"))
```

`List trailer` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

A list of warnings given by a server or intermediate proxy. Each warning is a 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` [HTTP Header]

A list of allowed methods on a given resource, as symbols.

```
(parse-header 'allow "GET, HEAD")
⇒ (GET HEAD)
```

`List content-encoding` [HTTP Header]

A list of content codings, as symbols.

```
(parse-header 'content-encoding "gzip")
⇒ (gzip)
```

`List content-language` [HTTP Header]

The languages that a resource is in, as strings.

```
(parse-header 'content-language "en")
⇒ ("en")
```

`UInt 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-length "text/plain")
⇒ (text/plain)
(parse-header 'content-length "text/plain;charset=utf-8")
⇒ (text/plain (charset . "utf-8"))
```

Note that the `charset` parameter is something is 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` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

A quality list of acceptable content codings.

```
(parse-header 'accept-encoding "gzip,identity=0.8")
⇒ ((1000 . "gzip") (800 . "identity"))
```

`QList accept-language` [HTTP Header]

A quality list of acceptable languages.

```
(parse-header 'accept-language "cn,en=0.75")
⇒ ((1000 . "cn") (750 . "en"))
```

`Pair authorization` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

The email address of a user making an HTTP request.

```
(parse-header 'from "bob@example.com")
⇒ "bob@example.com"
```

`Pair host` [HTTP Header]

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` [HTTP Header]

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.

```
(parse-header 'if-match "*")
⇒ *
(parse-header 'if-match "asdfadf")
⇒ (("asdfadf" . #t))
```

```
(parse-header 'if-match W/"asdfadf")
⇒ (("asdfadf" . #f))
```

`Date if-modified-since` [HTTP Header]

Indicates that a response should proceed if and only if the resource has been modified since the given date.

```
(parse-header 'if-modified-since "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`*|List if-none-match` [HTTP Header]

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.

```
(parse-header 'if-none-match "*")
⇒ *
```

`ETag|Date if-range` [HTTP Header]

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.

```
(parse-header 'if-range "\"original-etag\"")
⇒ ("original-etag" . #t)
```

`Date if-unmodified-since` [HTTP Header]

Indicates that a response should proceed if and only if the resource has not been modified since the given date.

```
(parse-header 'if-not-modified-since "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`UInt max-forwards` [HTTP Header]

The maximum number of proxy or gateway hops that a request should be subject to.

```
(parse-header 'max-forwards "10")
⇒ 10
```

`Pair proxy-authorization` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

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` [HTTP Header]

A list of transfer codings, expressed as key-value lists. A common transfer coding is `trailers`.

```
(parse-header 'te "trailers")
⇒ ((trailers))
```

`String user-agent` [HTTP Header]

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.

```
(parse-header 'user-agent "Mozilla/5.0")
⇒ "Mozilla/5.0"
```

7.3.4.5 Response Headers

`List accept-ranges` [HTTP Header]

A list of range units that the server supports, as symbols.

```
(parse-header 'accept-ranges "bytes")
⇒ (bytes)
```

`UInt age` [HTTP Header]

The age of a cached response, in seconds.

```
(parse-header 'age "3600")
⇒ 3600
```

`ETag etag` [HTTP Header]

The entity-tag of the resource.

```
(parse-header 'etag "\"foo\"")
⇒ ("foo" . #t)
```

`URI location` [HTTP Header]

A URI on which a request may be completed. Used in combination with a redirecting status code to perform client-side redirection.

```
(parse-header 'location "http://example.com/other")
⇒ #<uri ...>
```

`List proxy-authenticate` [HTTP Header]

A list of challenges to a proxy, indicating the need for authentication.

```
(parse-header 'proxy-authenticate "Basic realm=\"foo\"")
⇒ ((basic (realm . "foo")))
```

`UInt|Date retry-after` [HTTP Header]

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.

```
(parse-header 'retry-after "60")
⇒ 60
```

`String server` [HTTP Header]

A string identifying the server.

```
(parse-header 'server "My first web server")
⇒ "My first web server"
```

`*|List vary` [HTTP Header]

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.

```
(parse-header 'vary "Accept-Language, Accept")
⇒ (accept-language accept)
```

`List www-authenticate` [HTTP Header]

A list of challenges to a user, indicating the need for authentication.

```
(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.

```
(use-modules (web http))
```

`make-chunked-input-port` *port* [*#:keep-alive?=#f*] [Scheme Procedure]

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\r\n line\n Sec\r\n8\r\nond line\r\n0\r\n")
(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*] [Scheme Procedure]

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\nsecond chunk\r\n0\r\n"
```

7.3.6 HTTP Requests

```
(use-modules (web request))
```

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?` [Scheme Procedure]

`request-method` [Scheme Procedure]

`request-uri` [Scheme Procedure]
`request-version` [Scheme Procedure]
`request-headers` [Scheme Procedure]
`request-meta` [Scheme Procedure]
`request-port` [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 493).

`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]
[*#:headers='()*] [*#:port=#f*] [*#:meta='()*] [*#:validate-headers?=#t*]

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.

`read-request-body` *r* [Scheme Procedure]

Reads the request body from *r*, as a bytevector. Return `#f` if there was no request body.

`write-request-body` *r bv* [Scheme Procedure]

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 509, for more information on the format of parsed headers.

`request-accept` *request* [*default='()*] [Scheme Procedure]
`request-accept-charset` *request* [*default='()*] [Scheme Procedure]
`request-accept-encoding` *request* [*default='()*] [Scheme Procedure]
`request-accept-language` *request* [*default='()*] [Scheme Procedure]
`request-allow` *request* [*default='()*] [Scheme Procedure]
`request-authorization` *request* [*default=#f*] [Scheme Procedure]
`request-cache-control` *request* [*default='()*] [Scheme Procedure]
`request-connection` *request* [*default='()*] [Scheme Procedure]
`request-content-encoding` *request* [*default='()*] [Scheme Procedure]
`request-content-language` *request* [*default='()*] [Scheme Procedure]
`request-content-length` *request* [*default=#f*] [Scheme Procedure]
`request-content-location` *request* [*default=#f*] [Scheme Procedure]
`request-content-md5` *request* [*default=#f*] [Scheme Procedure]
`request-content-range` *request* [*default=#f*] [Scheme Procedure]
`request-content-type` *request* [*default=#f*] [Scheme Procedure]
`request-date` *request* [*default=#f*] [Scheme Procedure]
`request-expect` *request* [*default='()*] [Scheme Procedure]
`request-expires` *request* [*default=#f*] [Scheme Procedure]
`request-from` *request* [*default=#f*] [Scheme Procedure]
`request-host` *request* [*default=#f*] [Scheme Procedure]
`request-if-match` *request* [*default=#f*] [Scheme Procedure]
`request-if-modified-since` *request* [*default=#f*] [Scheme Procedure]
`request-if-none-match` *request* [*default=#f*] [Scheme Procedure]
`request-if-range` *request* [*default=#f*] [Scheme Procedure]
`request-if-unmodified-since` *request* [*default=#f*] [Scheme Procedure]
`request-last-modified` *request* [*default=#f*] [Scheme Procedure]
`request-max-forwards` *request* [*default=#f*] [Scheme Procedure]
`request-pragma` *request* [*default='()*] [Scheme Procedure]
`request-proxy-authorization` *request* [*default=#f*] [Scheme Procedure]
`request-range` *request* [*default=#f*] [Scheme Procedure]
`request-referer` *request* [*default=#f*] [Scheme Procedure]
`request-te` *request* [*default=#f*] [Scheme Procedure]
`request-trailer` *request* [*default='()*] [Scheme Procedure]
`request-transfer-encoding` *request* [*default='()*] [Scheme Procedure]
`request-upgrade` *request* [*default='()*] [Scheme Procedure]
`request-user-agent` *request* [*default=#f*] [Scheme Procedure]
`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*] [*default-port=#f*] [Scheme Procedure]

A helper routine to determine the absolute URI of a request, using the `host` header and the default host and port.

7.3.7 HTTP Responses

```
(use-modules (web response))
```

As with requests (see Section 7.3.6 [Requests], page 517), Guile offers a data type for HTTP responses. Again, the body is represented separately from the request.

`response?` [Scheme Procedure]
`response-version` [Scheme Procedure]
`response-code` [Scheme Procedure]
`response-reason-phrase` *response* [Scheme Procedure]
`response-headers` [Scheme Procedure]
`response-port` [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 519, for more information.

`build-response` [*#:version='(1 . 1)*] [*#:code=200*] [Scheme Procedure]
[*#: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* [Scheme Procedure]

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* [Scheme Procedure]

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.

`read-response-body` *r* [Scheme Procedure]

Read the response body from *r*, as a bytevector. Returns `#f` if there was no response body.

`write-response-body` *r bv* [Scheme Procedure]

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 509, for more information on the format of parsed headers.

`response-accept-ranges` *response* [*default=#f*] [Scheme Procedure]
`response-age` *response* [*default='()*] [Scheme Procedure]
`response-allow` *response* [*default='()*] [Scheme Procedure]
`response-cache-control` *response* [*default='()*] [Scheme Procedure]
`response-connection` *response* [*default='()*] [Scheme Procedure]
`response-content-encoding` *response* [*default='()*] [Scheme Procedure]
`response-content-language` *response* [*default='()*] [Scheme Procedure]
`response-content-length` *response* [*default=#f*] [Scheme Procedure]
`response-content-location` *response* [*default=#f*] [Scheme Procedure]
`response-content-md5` *response* [*default=#f*] [Scheme Procedure]
`response-content-range` *response* [*default=#f*] [Scheme Procedure]
`response-content-type` *response* [*default=#f*] [Scheme Procedure]
`response-date` *response* [*default=#f*] [Scheme Procedure]
`response-etag` *response* [*default=#f*] [Scheme Procedure]
`response-expires` *response* [*default=#f*] [Scheme Procedure]
`response-last-modified` *response* [*default=#f*] [Scheme Procedure]
`response-location` *response* [*default=#f*] [Scheme Procedure]
`response-pragma` *response* [*default='()*] [Scheme Procedure]
`response-proxy-authenticate` *response* [*default=#f*] [Scheme Procedure]
`response-retry-after` *response* [*default=#f*] [Scheme Procedure]
`response-server` *response* [*default=#f*] [Scheme Procedure]
`response-trailer` *response* [*default='()*] [Scheme Procedure]
`response-transfer-encoding` *response* [*default='()*] [Scheme Procedure]
`response-upgrade` *response* [*default='()*] [Scheme Procedure]
`response-vary` *response* [*default='()*] [Scheme Procedure]
`response-via` *response* [*default='()*] [Scheme Procedure]
`response-warning` *response* [*default='()*] [Scheme Procedure]
`response-www-authenticate` *response* [*default=#f*] [Scheme Procedure]

Return the given response header, or *default* if none was present.

7.3.8 Web Client

`(web client)` provides a simple, synchronous HTTP client, built on the lower-level HTTP, request, and response modules.

`open-socket-for-uri` *uri* [Scheme Procedure]

`http-get` *uri* [*#:port=(open-socket-for-uri uri)*] [*#:version='(1 . 1)*] [*#:keep-alive?=#f*] [*#:extra-headers='()*] [*#:decode-body?=#t*] [Scheme Procedure]

Connect to the server corresponding to *uri* and ask for the resource, using the `GET` method. If you already have a port open, pass it as *port*. The port will be closed at the end of the request unless *keep-alive?* is true. Any extra headers in the alist *extra-headers* will be added to the request.

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.

`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 522).

Another option, good but not as performant, would be to use threads, possibly via par-map or futures.

More helper procedures for the other common HTTP verbs would be a good addition to this module. Send your code to `guile-user@gnu.org`.

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 0 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 respose 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:

`define-server-impl` *name open read write close* [Scheme Procedure]

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` *handler* [*impl='http*] [*open-params='()*] . *state* [Scheme Procedure]

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 525.

The response and body will be run through `sanitize-response` before sending back to the client.

Additional arguments to *handler* are taken from *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` [*#:host=#f*] [*#:family=AF_INET*] [*#:addr=INADDR_LOOPBACK*] [*#:port 8080*] [*#:socket*] [HTTP Implementation]

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:

```
(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:

```
(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 8.4 [sxml simple], page 669).

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>\n")
                  (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
                (begin
                  (if doctype (display doctype port))
                  (sxml->xml sxml port))))))
```

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.

```
(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)` 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:

```
(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:

```
(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:

```
(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.

```
(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 "\
getopt-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 6.7.11 [Association Lists], page 220) 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 a 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:

```
$ ./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.

```
$ ./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:

```
$ ./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:

```
$ ./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

```
$ ./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* [Scheme Procedure]
: 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 533) 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-6 (see Section 7.5.6 [SRFI-6], page 558)). 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-6 bindings. If you want, you can do that already. We have included the module `(srfi srfi-6)` 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 program can of course directly use the various documented parts of Guile.

`cond-expand` (*feature body*...) ... [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.

```
(cond-expand (srfi-1
              )
             (else
              (define (alist-cons key val alist)
                (cons (cons key val) alist))))
```

Or demand a certain set of SRFIs (list operations, string ports, `receive` and string operations), failing if they're not available.

```
(cond-expand ((and srfi-1 srfi-6 srfi-8 srfi-13)
              ))
```

The Guile core has the following features,

```
guile
guile-2  ;; starting from Guile 2.x
r5rs
srfi-0
srfi-4
srfi-6
srfi-13
srfi-14
```

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.

```
(cond-expand (srfi-8
              )
             (guile
              (use-modules (srfi srfi-8))))
```

Likewise, testing the `guile-2` feature allows code to be portable between Guile 2.0 and previous versions of Guile. For instance, it makes it possible to write code that accounts for Guile 2.0's compiler, yet be correctly interpreted on 1.8 and earlier versions:

```
(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 6.22.2 [Feature Tracking], page 414), 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 6.7.2 [Lists], page 185 and Section 6.7.1 [Pairs], page 182.

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 procedure *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 6.7.2.3 [List Constructors], page 186) 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 6.7.2.2 [List Predicates], page 185).

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 `cdr`s 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* [Scheme Procedure]

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* [Scheme Procedure]

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* [Scheme Procedure]

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* ... [Scheme Procedure]

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* [Scheme Procedure]
`second` *pair* [Scheme Procedure]
`third` *pair* [Scheme Procedure]
`fourth` *pair* [Scheme Procedure]
`fifth` *pair* [Scheme Procedure]
`sixth` *pair* [Scheme Procedure]
`seventh` *pair* [Scheme Procedure]
`eighth` *pair* [Scheme Procedure]

`ninth` *pair* [Scheme Procedure]
`tenth` *pair* [Scheme Procedure]

These are synonyms for `car`, `cadr`, `caddr`, ...

`car+cdr` *pair* [Scheme Procedure]

Return two values, the CAR and the CDR of *pair*.

`take` *lst i* [Scheme Procedure]
`take!` *lst i* [Scheme Procedure]

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.

`drop` *lst i* [Scheme Procedure]

Return a list containing all but the first *i* elements of *lst*.

`take-right` *lst i* [Scheme Procedure]

Return a list containing the *i* last elements of *lst*. The return shares a common tail with *lst*.

`drop-right` *lst i* [Scheme Procedure]
`drop-right!` *lst i* [Scheme Procedure]

Return a list containing all but the *i* last elements of *lst*.

`drop-right` always returns a new list, even when *i* is zero. `drop-right!` may modify the structure of the argument list *lst* in order to produce the result.

`split-at` *lst i* [Scheme Procedure]
`split-at!` *lst i* [Scheme Procedure]

Return two values, a list containing the first *i* elements of the list *lst* and a list containing the remaining elements.

`split-at!` may modify the structure of the argument list *lst* in order to produce the result.

`last` *lst* [Scheme Procedure]

Return the last element of the non-empty, finite list *lst*.

7.5.3.4 Length, Append, Concatenate, etc.

`length+` *lst* [Scheme Procedure]

Return the length of the argument list *lst*. When *lst* is a circular list, `#f` is returned.

`concatenate` *list-of-lists* [Scheme Procedure]
`concatenate!` *list-of-lists* [Scheme Procedure]

Construct a list by appending all lists in *list-of-lists*.

`concatenate!` may modify the structure of the given lists in order to produce the result.

`concatenate` is the same as `(apply append list-of-lists)`. It exists because some Scheme implementations have a limit on the number of arguments a function takes, which the `apply` might exceed. In Guile there is no such limit.

`append-reverse` *rev-head tail* [Scheme Procedure]
`append-reverse!` *rev-head tail* [Scheme Procedure]

Reverse *rev-head*, append *tail* to it, and return the result. This is equivalent to `(append (reverse rev-head) tail)`, but its implementation is more efficient.

```
(append-reverse '(1 2 3) '(4 5 6)) ⇒ (3 2 1 4 5 6)
```

`append-reverse!` may modify *rev-head* in order to produce the result.

`zip` *lst1 lst2 ...* [Scheme Procedure]

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* [Scheme Procedure]
`unzip2` *lst* [Scheme Procedure]
`unzip3` *lst* [Scheme Procedure]
`unzip4` *lst* [Scheme Procedure]
`unzip5` *lst* [Scheme Procedure]

`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 ...* [Scheme Procedure]

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 ...* [Scheme Procedure]
`fold-right` *proc init lst1 lst2 ...* [Scheme Procedure]

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 *lst*s. At least one *lst* must be non-circular.

`fold` should be preferred over `fold-right` if the order of processing doesn't matter, or can be arranged either way, since `fold` is a little more efficient.

The way `fold` builds a result from iterating is quite general, it can do more than other iterations like say `map` or `filter`. The following for example removes adjacent duplicate elements from a list,

```
(define (delete-adjacent-duplicates lst)
  (fold-right (lambda (elem ret)
                (if (equal? elem (first ret))
                    ret
                    (cons elem ret)))
              (list (last lst))
              lst))
(delete-adjacent-duplicates '(1 2 3 3 4 4 4 5))
⇒ (1 2 3 4 5)
```

Clearly the same sort of thing can be done with a `for-each` and a variable in which to build the result, but a self-contained *proc* can be re-used in multiple contexts, where a `for-each` would have to be written out each time.

`pair-fold` *proc init lst1 lst2* ... [Scheme Procedure]
`pair-fold-right` *proc init lst1 lst2* ... [Scheme Procedure]

The same as `fold` and `fold-right`, but apply *proc* to the pairs of the lists instead of the list elements.

`reduce` *proc default lst* [Scheme Procedure]
`reduce-right` *proc default lst* [Scheme Procedure]

`reduce` is a variant of `fold`, where the first call to *proc* is on two elements from *lst*, rather than one element and a given initial value.

If *lst* is empty, `reduce` returns *default* (this is the only use for *default*). If *lst* has just one element then that's the return value. Otherwise *proc* is called on the elements of *lst*.

Each *proc* call is `(proc elem previous)`, where *elem* is from *lst* (the second and subsequent elements of *lst*), and *previous* is the return from the previous call to *proc*. The first element of *lst* is the *previous* for the first call to *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 `apply`.)

```
(reduce + 0 '(5 6 7)) ⇒ 18

(+ 6 5)  ⇒ 11
(+ 7 11) ⇒ 18
```

`reduce` can be used instead of `fold` where the *init* value is an "identity", meaning a value which under *proc* doesn't change the result, in this case 0 is an identity since `(+ 5 0)` is just 5. `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:

```
(unfold p f g seed) =
   (if (p seed) (tail-gen seed)
       (cons (f seed)
             (unfold p f g (g seed))))
```

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.

```
(let lp ((seed seed) (lis tail))
   (if (p seed) lis
       (lp (g seed)
           (cons (f seed) lis))))
```

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) '())`.

`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.

`append-map` *f lst1 lst2 ...* [Scheme Procedure]
`append-map!` *f lst1 lst2 ...* [Scheme Procedure]

Equivalent to

```
(apply append (map f clist1 clist2 ...))
```

and

```
(apply append! (map f clist1 clist2 ...))
```

Map *f* over the elements of the lists, just as in the `map` function. However, the results of the applications are appended together to make the final result. `append-map` uses `append` to append the results together; `append-map!` uses `append!`.

The dynamic order in which the various applications of *f* are made is not specified.

`map!` *f lst1 lst2 ...* [Scheme Procedure]

Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of *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, *lst2*, *lst3*, ... must have at least as many elements as *lst1*.

`pair-for-each` *f lst1 lst2 ...* [Scheme Procedure]

Like `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.

`filter-map` *f lst1 lst2 ...* [Scheme Procedure]

Like `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 `filter` and `filter!` functions are implemented in the Guile core, See Section 6.7.2.6 [List Modification], page 188.

`partition` *pred lst* [Scheme Procedure]
`partition!` *pred lst* [Scheme Procedure]

Split *lst* into those elements which do and don't satisfy the predicate *pred*.

The return is two values (see Section 6.13.7 [Multiple Values], page 290), the first being a list of all elements from *lst* which satisfy *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 `(`*pred* `elem)` are made on the list elements is unspecified.

`partition` does not change *lst*, but one of the returned lists may share a tail with it. `partition!` may modify *lst* to construct its return.

`remove` *pred lst* [Scheme Procedure]
`remove!` *pred lst* [Scheme Procedure]

Return a list containing all elements from *lst* which do not 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.

`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* [Scheme Procedure]

Return the first element of *lst* which satisfies the predicate *pred* and `#f` if no such element is found.

`find-tail` *pred lst* [Scheme Procedure]

Return the first pair of *lst* whose CAR satisfies the predicate *pred* and `#f` if no such element is found.

`take-while` *pred lst* [Scheme Procedure]
`take-while!` *pred lst* [Scheme Procedure]

Return 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* [Scheme Procedure]

Drop the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`span` *pred lst* [Scheme Procedure]
`span!` *pred lst* [Scheme Procedure]
`break` *pred lst* [Scheme Procedure]
`break!` *pred lst* [Scheme Procedure]

`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 283). 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 ...* [Scheme Procedure]

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 ...* [Scheme Procedure]

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 ...* [Scheme Procedure]

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* [=] [Scheme Procedure]

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 6.7.2.7 [List Searching], page 189) by accepting an equality predicate.

7.5.3.8 Deleting

`delete` *x lst* [=] [Scheme Procedure]
`delete!` *x lst* [=] [Scheme Procedure]

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 6.7.2.6 [List Modification], page 188) in accepting an equality predicate. See also `lset-difference` (see Section 7.5.3.10 [SRFI-1 Set Operations], page 545) 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 6.7.11 [Association Lists], page 220. 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 6.7.11.3 [Retrieving Alist Entries], page 222) by taking an optional = comparison procedure.

The default comparison is `equal?`. If an = parameter is given it's called `(= key alistcar)`, 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 6.7.11.2 [Adding or Setting Alist Entries], page 220) 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 alistkey)`, 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 6.7.4 [Bit Vectors], page 194) or hash tables (see Section 6.7.13 [Hash Tables], page 228) 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 265) 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)`.

```
(lset<= eq?)                        ⇒ #t
(lset<= eqv? '(1 2 3) '(1))         ⇒ #f
(lset<= eqv? '(1 3 2) '(4 3 1 2)) ⇒ #t
```

`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.

```
(lset= eq?)                       ⇒ #t
(lset= eqv? '(1 2 3) '(3 2 1))    ⇒ #t
(lset= string-ci=? '("a" "A" "b") '("B" "b" "a")) ⇒ #t
```

`lset-adjoin` = *list elem* ... [Scheme Procedure]

Add to *list* any of the given *elem*s not already in the list. *elem*s are `cons`ed onto the start of *list* (so the return value shares a common tail with *list*), but the order that the *elem*s 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 `cons`ed 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 290), 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))
```

`and-let*` (*clause* ...) *body* ... [library syntax]

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 (1+ 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]
`scm_make_s8vector` (*n*, *value*) [C Function]
`scm_make_u16vector` (*n*, *value*) [C Function]
`scm_make_s16vector` (*n*, *value*) [C Function]
`scm_make_u32vector` (*n*, *value*) [C Function]
`scm_make_s32vector` (*n*, *value*) [C Function]
`scm_make_u64vector` (*n*, *value*) [C Function]
`scm_make_s64vector` (*n*, *value*) [C Function]
`scm_make_f32vector` (*n*, *value*) [C Function]
`scm_make_f64vector` (*n*, *value*) [C Function]
`scm_make_c32vector` (*n*, *value*) [C Function]
`scm_make_c64vector` (*n*, *value*) [C Function]

Return a newly allocated homogeneous numeric vector holding *n* elements of the indicated type. If *value* is given, the vector is initialized with that value, otherwise the contents are unspecified.

`u8vector` *value* ... [Scheme Procedure]
`s8vector` *value* ... [Scheme Procedure]
`u16vector` *value* ... [Scheme Procedure]
`s16vector` *value* ... [Scheme Procedure]
`u32vector` *value* ... [Scheme Procedure]
`s32vector` *value* ... [Scheme Procedure]
`u64vector` *value* ... [Scheme Procedure]
`s64vector` *value* ... [Scheme Procedure]
`f32vector` *value* ... [Scheme Procedure]
`f64vector` *value* ... [Scheme Procedure]
`c32vector` *value* ... [Scheme Procedure]
`c64vector` *value* ... [Scheme Procedure]
`scm_u8vector` (*values*) [C Function]
`scm_s8vector` (*values*) [C Function]
`scm_u16vector` (*values*) [C Function]
`scm_s16vector` (*values*) [C Function]
`scm_u32vector` (*values*) [C Function]

`scm_s32vector` (*values*) [C Function]
`scm_u64vector` (*values*) [C Function]
`scm_s64vector` (*values*) [C Function]
`scm_f32vector` (*values*) [C Function]
`scm_f64vector` (*values*) [C Function]
`scm_c32vector` (*values*) [C Function]
`scm_c64vector` (*values*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, holding the given parameter *value*s. The vector length is the number of parameters given.

`u8vector-length` *vec* [Scheme Procedure]
`s8vector-length` *vec* [Scheme Procedure]
`u16vector-length` *vec* [Scheme Procedure]
`s16vector-length` *vec* [Scheme Procedure]
`u32vector-length` *vec* [Scheme Procedure]
`s32vector-length` *vec* [Scheme Procedure]
`u64vector-length` *vec* [Scheme Procedure]
`s64vector-length` *vec* [Scheme Procedure]
`f32vector-length` *vec* [Scheme Procedure]
`f64vector-length` *vec* [Scheme Procedure]
`c32vector-length` *vec* [Scheme Procedure]
`c64vector-length` *vec* [Scheme Procedure]
`scm_u8vector_length` (*vec*) [C Function]
`scm_s8vector_length` (*vec*) [C Function]
`scm_u16vector_length` (*vec*) [C Function]
`scm_s16vector_length` (*vec*) [C Function]
`scm_u32vector_length` (*vec*) [C Function]
`scm_s32vector_length` (*vec*) [C Function]
`scm_u64vector_length` (*vec*) [C Function]
`scm_s64vector_length` (*vec*) [C Function]
`scm_f32vector_length` (*vec*) [C Function]
`scm_f64vector_length` (*vec*) [C Function]
`scm_c32vector_length` (*vec*) [C Function]
`scm_c64vector_length` (*vec*) [C Function]

Return the number of elements in *vec*.

`u8vector-ref` *vec i* [Scheme Procedure]
`s8vector-ref` *vec i* [Scheme Procedure]
`u16vector-ref` *vec i* [Scheme Procedure]
`s16vector-ref` *vec i* [Scheme Procedure]
`u32vector-ref` *vec i* [Scheme Procedure]
`s32vector-ref` *vec i* [Scheme Procedure]
`u64vector-ref` *vec i* [Scheme Procedure]
`s64vector-ref` *vec i* [Scheme Procedure]
`f32vector-ref` *vec i* [Scheme Procedure]
`f64vector-ref` *vec i* [Scheme Procedure]
`c32vector-ref` *vec i* [Scheme Procedure]
`c64vector-ref` *vec i* [Scheme Procedure]

`scm_u8vector_ref` (*vec*, *i*) [C Function]
`scm_s8vector_ref` (*vec*, *i*) [C Function]
`scm_u16vector_ref` (*vec*, *i*) [C Function]
`scm_s16vector_ref` (*vec*, *i*) [C Function]
`scm_u32vector_ref` (*vec*, *i*) [C Function]
`scm_s32vector_ref` (*vec*, *i*) [C Function]
`scm_u64vector_ref` (*vec*, *i*) [C Function]
`scm_s64vector_ref` (*vec*, *i*) [C Function]
`scm_f32vector_ref` (*vec*, *i*) [C Function]
`scm_f64vector_ref` (*vec*, *i*) [C Function]
`scm_c32vector_ref` (*vec*, *i*) [C Function]
`scm_c64vector_ref` (*vec*, *i*) [C Function]

Return the element at index *i* in *vec*. The first element in *vec* is index 0.

`u8vector-set!` *vec i value* [Scheme Procedure]
`s8vector-set!` *vec i value* [Scheme Procedure]
`u16vector-set!` *vec i value* [Scheme Procedure]
`s16vector-set!` *vec i value* [Scheme Procedure]
`u32vector-set!` *vec i value* [Scheme Procedure]
`s32vector-set!` *vec i value* [Scheme Procedure]
`u64vector-set!` *vec i value* [Scheme Procedure]
`s64vector-set!` *vec i value* [Scheme Procedure]
`f32vector-set!` *vec i value* [Scheme Procedure]
`f64vector-set!` *vec i value* [Scheme Procedure]
`c32vector-set!` *vec i value* [Scheme Procedure]
`c64vector-set!` *vec i value* [Scheme Procedure]
`scm_u8vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s8vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u16vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s16vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_f32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_f64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_c32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_c64vector_set_x` (*vec*, *i*, *value*) [C Function]

Set the element at index *i* in *vec* to *value*. The first element in *vec* is index 0. The return value is unspecified.

`u8vector->list` *vec* [Scheme Procedure]
`s8vector->list` *vec* [Scheme Procedure]
`u16vector->list` *vec* [Scheme Procedure]
`s16vector->list` *vec* [Scheme Procedure]
`u32vector->list` *vec* [Scheme Procedure]
`s32vector->list` *vec* [Scheme Procedure]
`u64vector->list` *vec* [Scheme Procedure]

`s64vector->list` *vec* [Scheme Procedure]
`f32vector->list` *vec* [Scheme Procedure]
`f64vector->list` *vec* [Scheme Procedure]
`c32vector->list` *vec* [Scheme Procedure]
`c64vector->list` *vec* [Scheme Procedure]
`scm_u8vector_to_list` (*vec*) [C Function]
`scm_s8vector_to_list` (*vec*) [C Function]
`scm_u16vector_to_list` (*vec*) [C Function]
`scm_s16vector_to_list` (*vec*) [C Function]
`scm_u32vector_to_list` (*vec*) [C Function]
`scm_s32vector_to_list` (*vec*) [C Function]
`scm_u64vector_to_list` (*vec*) [C Function]
`scm_s64vector_to_list` (*vec*) [C Function]
`scm_f32vector_to_list` (*vec*) [C Function]
`scm_f64vector_to_list` (*vec*) [C Function]
`scm_c32vector_to_list` (*vec*) [C Function]
`scm_c64vector_to_list` (*vec*) [C Function]

Return a newly allocated list holding all elements of *vec*.

`list->u8vector` *lst* [Scheme Procedure]
`list->s8vector` *lst* [Scheme Procedure]
`list->u16vector` *lst* [Scheme Procedure]
`list->s16vector` *lst* [Scheme Procedure]
`list->u32vector` *lst* [Scheme Procedure]
`list->s32vector` *lst* [Scheme Procedure]
`list->u64vector` *lst* [Scheme Procedure]
`list->s64vector` *lst* [Scheme Procedure]
`list->f32vector` *lst* [Scheme Procedure]
`list->f64vector` *lst* [Scheme Procedure]
`list->c32vector` *lst* [Scheme Procedure]
`list->c64vector` *lst* [Scheme Procedure]
`scm_list_to_u8vector` (*lst*) [C Function]
`scm_list_to_s8vector` (*lst*) [C Function]
`scm_list_to_u16vector` (*lst*) [C Function]
`scm_list_to_s16vector` (*lst*) [C Function]
`scm_list_to_u32vector` (*lst*) [C Function]
`scm_list_to_s32vector` (*lst*) [C Function]
`scm_list_to_u64vector` (*lst*) [C Function]
`scm_list_to_s64vector` (*lst*) [C Function]
`scm_list_to_f32vector` (*lst*) [C Function]
`scm_list_to_f64vector` (*lst*) [C Function]
`scm_list_to_c32vector` (*lst*) [C Function]
`scm_list_to_c64vector` (*lst*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, initialized with the elements of the list *lst*.

`SCM scm_take_u8vector` (*const scm_t_uint8 *data, size_t len*) [C Function]
`SCM scm_take_s8vector` (*const scm_t_int8 *data, size_t len*) [C Function]

SCM scm_take_u16vector (*const scm_t_uint16 *data, size_t len*) [C Function]
SCM scm_take_s16vector (*const scm_t_int16 *data, size_t len*) [C Function]
SCM scm_take_u32vector (*const scm_t_uint32 *data, size_t len*) [C Function]
SCM scm_take_s32vector (*const scm_t_int32 *data, size_t len*) [C Function]
SCM scm_take_u64vector (*const scm_t_uint64 *data, size_t len*) [C Function]
SCM scm_take_s64vector (*const scm_t_int64 *data, size_t len*) [C Function]
SCM scm_take_f32vector (*const float *data, size_t len*) [C Function]
SCM scm_take_f64vector (*const double *data, size_t len*) [C Function]
SCM scm_take_c32vector (*const float *data, size_t len*) [C Function]
SCM scm_take_c64vector (*const double *data, size_t len*) [C Function]

Return a new uniform numeric vector of the indicated type and length that uses the memory pointed to by *data* to store its elements. This memory will eventually be freed with `free`. The argument *len* specifies the number of elements in *data*, not its size in bytes.

The `c32` and `c64` variants take a pointer to a C array of `float`s or `double`s. The real parts of the complex numbers are at even indices in that array, the corresponding imaginary parts are at the following odd index.

const scm_t_uint8 * scm_u8vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_int8 * scm_s8vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_uint16 * scm_u16vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_int16 * scm_s16vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_uint32 * scm_u32vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_int32 * scm_s32vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_uint64 * scm_u64vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const scm_t_int64 * scm_s64vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const float * scm_f32vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const double * scm_f64vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const float * scm_c32vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
const double * scm_c64vector_elements (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` (see Section 6.7.3.4 [Vector Accessing from C], page 193), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

`scm_t_uint8 * scm_u8vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_int8 * scm_s8vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_uint16 * scm_u16vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_int16 * scm_s16vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_uint32 * scm_u32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_int32 * scm_s32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_uint64 * scm_u64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`scm_t_int64 * scm_s64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`float * scm_f32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`double * scm_f64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`float * scm_c32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
`double * scm_c64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_writable_elements` (see Section 6.7.3.4 [Vector Accessing from C], page 193), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

7.5.5.3 SRFI-4 - Generic operations

Guile also provides procedures that operate on all types of uniform numeric vectors. In what is probably a bug, these procedures are currently available in the default environment as well; however prudent hackers will make sure to import `(srfi srfi-4 gnu)` before using these.

`int scm_is_uniform_vector` (*SCM uvec*) [C Function]

Return non-zero when *uvec* is a uniform numeric vector, zero otherwise.

`size_t scm_c_uniform_vector_length` (*SCM uvec*) [C Function]

Return the number of elements of *uvec* as a `size_t`.

`uniform-vector?` *obj* [Scheme Procedure]
`scm_uniform_vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a homogeneous numeric vector of the indicated type.

`uniform-vector-length` *vec* [Scheme Procedure]
`scm_uniform_vector_length` (*vec*) [C Function]

Return the number of elements in *vec*.

`uniform-vector-ref` *vec i* [Scheme Procedure]
`scm_uniform_vector_ref` (*vec, i*) [C Function]
Return the element at index *i* in *vec*. The first element in *vec* is index 0.

`uniform-vector-set!` *vec i value* [Scheme Procedure]
`scm_uniform_vector_set_x` (*vec, i, value*) [C Function]
Set the element at index *i* in *vec* to *value*. The first element in *vec* is index 0. The return value is unspecified.

`uniform-vector->list` *vec* [Scheme Procedure]
`scm_uniform_vector_to_list` (*vec*) [C Function]
Return a newly allocated list holding all elements of *vec*.

`const void * scm_uniform_vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
Like `scm_vector_elements` (see Section 6.7.3.4 [Vector Accessing from C], page 193), but returns a pointer to the elements of a uniform numeric vector.

`void * scm_uniform_vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]
Like `scm_vector_writable_elements` (see Section 6.7.3.4 [Vector Accessing from C], page 193), but returns a pointer to the elements of a uniform numeric vector.

Unless you really need to the limited generality of these functions, it is best to use the type-specific functions, or the generalized vector accessors.

7.5.5.4 SRFI-4 - Relation to bytevectors

Guile implements SRFI-4 vectors using bytevectors (see Section 6.6.6 [Bytevectors], page 162). 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 6.6.6 [Bytevectors], page 162, for more information.

7.5.5.5 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:

```
(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]
`scm_any_to_s8vector` (*obj*) [C Function]
`scm_any_to_u16vector` (*obj*) [C Function]
`scm_any_to_s16vector` (*obj*) [C Function]
`scm_any_to_u32vector` (*obj*) [C Function]
`scm_any_to_s32vector` (*obj*) [C Function]
`scm_any_to_u64vector` (*obj*) [C Function]
`scm_any_to_s64vector` (*obj*) [C Function]
`scm_any_to_f32vector` (*obj*) [C Function]
`scm_any_to_f64vector` (*obj*) [C Function]
`scm_any_to_c32vector` (*obj*) [C Function]
`scm_any_to_c64vector` (*obj*) [C Function]

Return a (maybe newly allocated) uniform numeric vector of the indicated type, initialized with the elements of *obj*, which must be a list, a vector, or a uniform vector. When *obj* is already a suitable uniform numeric vector, it is returned unchanged.

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 290.

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. In Guile this is simply an alternate interface to the core record functionality (see Section 6.7.8 [Records], page 212). It can be used with,

```
(use-modules (srfi srfi-9))
```

`define-record-type` *type* [library syntax]
(*constructor fieldname* ...)
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 *fieldname*s 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,

```
(define-record-type employee-type
  (make-employee name age salary)
  employee?
  (name    get-employee-name)
  (age     get-employee-age    set-employee-age)
  (salary  get-employee-salary set-employee-salary))
```

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,

```
employee-type ⇒ #<record-type employee-type>

(define fred (make-employee "Fred" 45 20000.00))

(employee? fred)        ⇒ #t
(get-employee-age fred) ⇒ 45
(set-employee-salary fred 25000.00)  ;; pay rise
```

The functions created by `define-record-type` are ordinary top-level `define`s. They can be redefined or `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 `lambda` body or inside a *let* block. However, Guile's implementation does not enforce that restriction.

Custom Printers

You may use `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 `(srfi srfi-9 gnu)` module.

`set-record-type-printer!` *name thunk* [Scheme Syntax]

Where *type* corresponds to the first argument of `define-record-type`, and *thunk* 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]`.

```
(set-record-type-printer! employee-type
  (lambda (record port)
    (write-char #\[ port)
    (display (get-employee-name record) port)
    (write-char #\] port)))
```

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 *arg*s are objects taken as parameters. *tag*s 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.

```
(define-reader-ctor 'repeat
  (lambda (obj reps)
    (make-list reps obj)))

(display '#,(repeat 99 3))
⊣ (99 99 99)
```

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.

```
(display '#,(repeat 99 3))
⇒
(display '(99 99 99))
```

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,

```
(define-reader-ctor 'sum
  (lambda (x y)
    (+ x y)))
(display #,(sum 123 456)) ⊣ 579
```

A typical use for `#,()` is to get a read syntax for objects which don't otherwise have one. For example, the following allows a hash table to be given literally, with tags and values, ready for fast lookup.

```
(define-reader-ctor 'hash
  (lambda elems
    (let ((table (make-hash-table)))
      (for-each (lambda (elem)
                  (apply hash-set! table elem))
                elems)
      table)))

(define (animal->family animal)
  (hash-ref '#,(hash ("tiger" "cat")
                     ("lion"  "cat")
                     ("wolf"  "dog"))
            animal))

(animal->family "lion") ⇒ "cat"
```

Or for example the following is a syntax for a compiled regular expression (see Section 6.15 [Regular Expressions], page 335).

```
(use-modules (ice-9 regex))

(define-reader-ctor 'regexp make-regexp)

(define (extract-angs str)
  (let ((match (regexp-exec '#,(regexp "<([A-Z0-9]+)>") str)))
    (and match
         (match:substring match 1))))

(extract-angs "foo <BAR> quux") ⇒ "BAR"
```

`#,()` is somewhat similar to `define-macro` (see Section 6.10 [Macros], page 248) in that handler code is run to produce a result, but `#,()` operates at the read stage, so it can appear in data for `read` (see Section 6.17.2 [Scheme Read], page 346), not just in code to be executed.

Because `#,()` is handled at read-time it has no direct access to variables etc. A symbol in the arguments is just a symbol, not a variable reference. The arguments are essentially constants, though the handler procedure can use them in any complicated way it might want.

Once `(srfi srfi-10)` has loaded, `#,()` is available globally, there's no need to use `(srfi srfi-10)` in later modules. Similarly the tags registered are global and can be used anywhere once registered.

There's no attempt to record what previous `#,()` forms have been seen, if two identical forms occur then two calls are made to the handler procedure. The handler might like to maintain a cache or similar to avoid making copies of large objects, depending on expected usage.

In code the best uses of `#,()` are generally when there's a lot of objects of a particular kind as literals or constants. If there's just a few then some local variables and initializers are fine, but that becomes tedious and error prone when there's a lot, and the anonymous and compact syntax of `#,()` is much better.

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 277), but they support binding of the values returned by multiple-valued expressions.

Write `(use-modules (srfi srfi-11))` to make the bindings available.

```
(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 140.

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 133.

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 243, 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

```
(use-modules (srfi srfi-17))
```

For example `vector-ref` is extended so that

```
(set! (vector-ref vec idx) new-value)
```

is equivalent to

```
(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 [Procedures with Setters], page 246). This module adds definitions of the following functions as procedures with setters, allowing them to be targets of a `set!`,

`car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `caaaar`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cddaar`, `cddadr`, `cdddar`, `cddddr`

`string-ref`, `vector-ref`

The SRFI specifies `setter` (see Section 6.9.8 [Procedures with Setters], page 246) 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).

`getter-with-setter` [Function]

The same as the Guile core `make-procedure-with-setter` (see Section 6.9.8 [Procedures with Setters], page 246).

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

```
(use-modules (srfi srfi-18))
```

As a general rule, the data types and functions in this SRFI-18 implementation are compatible with the types and functions in Guile's core threading code. For example, mutexes created with the SRFI-18 `make-mutex` function can be passed to the built-in Guile function `lock-mutex` (see Section 6.21.4 [Mutexes and Condition Variables], page 400), and mutexes created with the built-in Guile function `make-mutex` can be passed to the SRFI-18 function `mutex-lock!`. Cases in which this does not hold true are noted in the following sections.

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. In all other regards, SRFI-18 threads are identical to normal Guile threads.

`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 6.21.3 [Threads], page 398).

`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 6.21.3 [Threads], page 398).

`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 566). 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 6.21.3 [Threads], page 398) 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 567). 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

The behavior of Guile's built-in mutexes is parameterized via a set of flags passed to the `make-mutex` procedure in the core (see Section 6.21.4 [Mutexes and Condition Variables], page 400). To satisfy the requirements for mutexes specified by SRFI-18, the `make-mutex` procedure described below sets the following flags:

- `recursive`: the mutex can be locked recursively
- `unchecked-unlock`: attempts to unlock a mutex that is already unlocked will not raise an exception
- `allow-external-unlock`: the mutex can be unlocked by any thread, not just the thread that locked it originally

`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. Note that the name `make-mutex` conflicts with Guile core function `make-mutex`. Applications wanting to use both of these functions will need to refer to them by different names.

`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]
`mutex-specific-set!` *mutex obj* [Function]

Get or set the "object-specific" property of *mutex*. In Guile's implementation of SRFI-18, this value is stored as an object property, and will be `#f` if not set.

`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. This procedure has the same behavior as the `lock-mutex` procedure in the core library.

`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. This procedure has the same behavior as the `unlock-mutex` procedure in the core library.

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, or Guile's built-in `wait-condition-variable` procedure can be used.

`condition-variable?` *obj* [Function]

Returns `#t` if *obj* is a condition variable, `#f` otherwise. This is the same procedure as the same-named built-in procedure (see Section 6.21.4 [Mutexes and Condition Variables], page 400).

`make-condition-variable` [*name*] [Function]

Returns a new condition variable, optionally assigning it the object name *name*, which may be any Scheme object. This procedure replaces a procedure of the same name in the core library.

`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]
`condition-variable-specific-set!` *condition-variable obj* [Function]

Get or set the "object-specific" property of *condition-variable*. In Guile's implementation of SRFI-18, this value is stored as an object property, and will be `#f` if not set.

`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!`. The behavior of these procedures is equivalent to that of the procedures `signal-condition-variable` and `broadcast-condition-variable` in the core library.

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]
`seconds->time` *seconds* [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* [Function]
`uncaught-exception-reason` *exc* [Function]

`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 exception 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))
```

Caution: The current code in this module incorrectly extends the Gregorian calendar leap year rule back prior to the introduction of those reforms in 1582 (or the appropriate year in various countries). The Julian calendar was used prior to 1582, and there were 10 days skipped for the reform, but the code doesn't implement that.

This will be fixed some time. Until then calculations for 1583 onwards are correct, but prior to that any day/month/year and day of the week calculations are wrong.

7.5.16.1 SRFI-19 Introduction

This module implements time and date representations and calculations, in various time systems, including universal time (UTC) and 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* is a real number which is a count of days and fraction of a day, in UTC, starting from -4713-01-01T12:00:00Z, ie. midday Monday 1 Jan 4713 B.C. A *Modified Julian Day* is the same, but starting from 1858-11-17T00:00:00Z, ie. midnight 17 November 1858 UTC. 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.

`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.

`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*.

`time-type` *time* [Function]
`time-nanosecond` *time* [Function]
`time-second` *time* [Function]
`set-time-type!` *time type* [Function]
`set-time-nanosecond!` *time nsec* [Function]
`set-time-second!` *time sec* [Function]

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 571.

`copy-time` *time* [Function]

Return a new time object, which is a copy of the given *time*.

`current-time` [*type*] [Function]

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 468) as well as the SRFI-18 `current-time` function (see Section 7.5.15.4 [SRFI-18 Time], page 566). Applications wanting to use more than one of these functions will need to refer to them by different names.

`time-resolution` [*type*] [Function]

Return the resolution, in nanoseconds, of the given time *type*. The default *type* is `time-utc`.

`time<=?` *t1 t2* [Function]
`time<?` *t1 t2* [Function]

`time=?` *t1 t2* [Function]
`time>=?` *t1 t2* [Function]
`time>?` *t1 t2* [Function]

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* [Function]
`time-difference!` *t1 t2* [Function]

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* [Function]
`add-duration!` *time duration* [Function]
`subtract-duration` *time duration* [Function]
`subtract-duration!` *time duration* [Function]

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.

`date?` *obj* [Function]

Return `#t` if *obj* is a date object, or `#f` if not.

`make-date` *nsecs seconds minutes hours date month year zone-offset* [Function]

Create a new date object.

`date-nanosecond` *date* [Function]

Nanoseconds, 0 to 999999999.

`date-second` *date* [Function]

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* [Function]

Minutes, 0 to 59.

`date-hour` *date* [Function]

Hour, 0 to 23.

`date-day` *date* [Function]

Day of the month, 1 to 31 (or less, according to the month).

`date-month` *date* [Function]
Month, 1 to 12.

`date-year` *date* [Function]
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* [Function]
Time zone, an integer number of seconds east of Greenwich.

`date-year-day` *date* [Function]
Day of the year, starting from 1 for 1st January.

`date-week-day` *date* [Function]
Day of the week, starting from 0 for Sunday.

`date-week-number` *date dstartw* [Function]
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*] [Function]
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` [Function]
Return the current Julian Day.

`current-modified-julian-day` [Function]
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]
`time-utc->modified-julian-day` *time* [Function]
`time-utc->time-monotonic` *time* [Function]
`time-utc->time-monotonic!` *time* [Function]
`time-utc->time-tai` *time* [Function]
`time-utc->time-tai!` *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 468).

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 468), 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, '~k:~M:~S~z'
~3	ISO-8601 time, '~k:~M:~S'
~4	ISO-8601 date/time+zone, '~Y-~m-~dT~k:~M:~S~z'
~5	ISO-8601 date/time, '~Y-~m-~dT~k:~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 6.24.5 [Accessing Locale Information], page 424). See Section 7.2.13 [Locales], page 501, 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.

	Skip to	Read	Set
`~~`	no skip	literal ~	nothing
`~a`	`char-alphabetic?`	locale abbreviated weekday name	nothing
`~A`	`char-alphabetic?`	locale full weekday name	nothing
`~b`	`char-alphabetic?`	locale abbreviated month name	`date-mont`
`~B`	`char-alphabetic?`	locale full month name	`date-mont`
`~d`	`char-numeric?`	day of month	`date-day`
`~e`	no skip	day of month, blank padded	`date-day`
`~h`	same as '`~b`'		
`~H`	`char-numeric?`	hour	`date-hour`
`~k`	no skip	hour, blank padded	`date-hour`
`~m`	`char-numeric?`	month	`date-mont`
`~M`	`char-numeric?`	minute	`date-minu`
`~S`	`char-numeric?`	second	`date-seco`
`~y`	no skip	2-digit year	`date-year` 50 years
`~Y`	`char-numeric?`	year	`date-year`
`~z`	no skip	time zone	date-zone-

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 6.24.5 [Accessing Locale Information], page 424). See Section 7.2.13 [Locales], page 501, 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 577 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* [Function]

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 an reinstated with the following procedures:

`random-source-state-ref` *source* [Function]
`random-source-state-set!` *source state* [Function]

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* [Function]

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* [Function]
`random-source-make-reals` *source unit* [Function]

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 * *unit* where x is a random integer in {1, ..., 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-30 - Nested Multi-line Comments

Starting from version 2.0, Guile's `read` supports SRFI-30/R6RS nested multi-line comments by default, Section 6.17.1.3 [Block Comments], page 345.

7.5.21 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))))
```

The second syntax can be used to create anonymous recursive functions:

```
guile> (define tmp (rec (display-n item n)
                     (if (positive? n)
                         (begin (display n) (display-n (- n 1))))))
guile> (tmp 42 3)
424242
guile>
```

7.5.22 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 291). It can be made available as follows:

```
(use-modules (srfi srfi-34))
```

7.5.23 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`:

```
(raise (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:

```
(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:

```
(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 *conditions*. The returned condition has the type of each condition of *conditions* (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.

7.5.24 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 528). 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 msg) (quit)))))
             (list (option '(#\v "version") #f #f
                           (display-and-exit-proc "Foo version 42.0\n"))
                   (option '(#\h "help") #f #f
                           (display-and-exit-proc
                            "Usage: foo scheme-file ...")))
           (lambda (opt name arg loads)
             (error "Unrecognized option `~A'" 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 `args-fold` "seeds", and the *processor* should return seeds as well.

`option-names` *opt* [Scheme Procedure]
`option-required-arg?` *opt* [Scheme Procedure]
`option-optional-arg?` *opt* [Scheme Procedure]
`option-processor` *opt* [Scheme Procedure]

Return the specified field of *opt*, an option object, as described above for `option`.

`args-fold` *args options unrecognized-option-proc operand-proc seed ...* [Scheme Procedure]

Process *args*, a list of program arguments such as that returned by `(cdr (program-arguments))`, in order against *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 *seed* ..., and must return the new seeds. Return the final seeds.

Call `unrecognized-option-proc`, which is like an option object's processor, for any options not found in *options*.

Call `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.

7.5.25 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 `write-with-shared-structure` and `read-with-shared-structure`. It is identical to the grammar for external representation for data written and read with `write` and `read` given in section 7 of R5RS, except that the single production

```
<datum> --> <simple datum> | <compound datum>
```

is replaced by the following five productions:

```
<datum> --> <defining datum> | <nondefining datum> | <defined datum>
<defining datum> -->  #<indexnum>=<nondefining datum>
<defined datum> --> #<indexnum>#
<nondefining datum> --> <simple datum> | <compound datum>
<indexnum> --> <digit 10>+
```

`write-with-shared-structure` *obj* [Scheme procedure]
`write-with-shared-structure` *obj port* [Scheme procedure]
`write-with-shared-structure` *obj port optarg* [Scheme procedure]

Writes an external representation of *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 `write-with-shared-structure` must produce the same external representation for those objects as `write`.

`write-with-shared-structure` terminates in finite time and produces a finite representation when writing finite data.

`write-with-shared-structure` returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `(current-output-port)`. The *optarg* argument may also be omitted. If present, its effects on the output and return value are unspecified but `write-with-shared-structure` must still write a representation that can be read by `read-with-shared-structure`. Some implementations may wish to use *optarg* to specify formatting conventions, numeric radixes, or return values. Guile's implementation ignores *optarg*.

For example, the code

```
(begin (define a (cons 'val1 'val2))
       (set-cdr! a a)
       (write-with-shared-structure a))
```

should produce the output `#1=(val1 . #1#)`. This shows a cons cell whose `cdr` contains itself.

`read-with-shared-structure` [Scheme procedure]
`read-with-shared-structure` *port* [Scheme procedure]

`read-with-shared-structure` converts the external representations of Scheme objects produced by `write-with-shared-structure` into Scheme objects. That is, it is a parser for the nonterminal '`<datum>`' in the augmented external representation grammar defined above. `read-with-shared-structure` returns the next object parsable from the given input port, updating *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 `read-with-shared-structure` or `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 *port* argument may be omitted, in which case it defaults to the value returned by `(current-input-port)`. It is an error to read from a closed port.

7.5.26 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 6.21.8 [Parameters], page 408.

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 6.21.7 [Fluids and Dynamic States], page 405).

`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*`.

7.5.27 SRFI-42 - Eager Comprehensions

See the specification of SRFI-42 (`http://srfi.schemers.org/srfi-42/srfi-42.html`).

7.5.28 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.

`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

```
(let ((value expression)) (delay value)).
```

However, the former is more efficient since it does not require unnecessary creation and evaluation of thunks. We also have the equivalence

```
(delay expression) = (lazy (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:

```
(force (delay expression)) -> expression
(force (lazy  expression)) -> (force expression)
(force (eager value))      -> value
```

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

```
(define (stream-filter p? s)
  (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:

```
(define (stream-filter p? s)
  (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 ...)`.

7.5.29 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` *clause1 clause2 . . .* [library syntax]

Require the features of *clause1 clause2 . . .* , throwing an error if any are unavailable.

A *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,

```
(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.

7.5.30 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 124). 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 ...* [Function]
`bitwise-ior` *n1 ...* [Function]
`bitwise-xor` *n1 ...* [Function]
`bitwise-not` *n* [Function]
`any-bits-set?` *j k* [Function]
`bit-set?` *index n* [Function]
`arithmetic-shift` *n count* [Function]
`bit-field` *n start end* [Function]
`bit-count` *n* [Function]

Aliases for `logand`, `logior`, `logxor`, `lognot`, `logtest`, `logbit?`, `ash`, `bit-extract` and `logcount` respectively.

Note that the name `bit-count` conflicts with `bit-count` in the core (see Section 6.7.4 [Bit Vectors], page 194).

`bitwise-if` *mask n1 n0* [Function]
`bitwise-merge` *mask n1 n0* [Function]

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* [Function]
`first-set-bit` *n* [Function]

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* [Function]

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* [Function]

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* [Function]

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 #b0110 2 1 4) ⇒ #b1010
```

`reverse-bit-field` *n start end* [Function]

Return *n* with the bits from *start* (inclusive) to *end* (exclusive) reversed.

```
(reverse-bit-field #b101001 2 4) ⇒ #b100101
```

`integer->list` *n* [*len*] [Function]

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 124).

```
(integer->list 6)   ⇒ (#t #t #f)
(integer->list 1 4) ⇒ (#f #f #f #t)
```

`list->integer` *lst* [Function]
`booleans->integer` *bool...* [Function]

Return an integer formed bitwise from the given *lst* list of booleans, 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
```

7.5.31 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 281.

7.5.32 SRFI-67 - Compare procedures

See the specification of SRFI-67 (`http://srfi.schemers.org/srfi-67/srfi-67.html`).

7.5.33 SRFI-69 - Basic hash tables

This is a portable wrapper around Guile's built-in hash table and weak table support. See Section 6.7.13 [Hash Tables], page 228, 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))
```

7.5.33.1 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 6.18.3 [Weak References], page 362, 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.

7.5.33.2 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".

7.5.33.3 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` *table proc init* [Scheme Procedure]

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` *table* [Scheme Procedure]

Answer an alist where each association in *table* is an association in the result.

7.5.33.4 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` *hash-table* [Scheme Procedure]
`hash-table-hash-function` *hash-table* [Scheme Procedure]

Answer the equivalence and hash function of *hash-table*, respectively.

`hash` *obj* [*size*] [Scheme Procedure]
`string-hash` *obj* [*size*] [Scheme Procedure]
`string-ci-hash` *obj* [*size*] [Scheme Procedure]
`hash-by-identity` *obj* [*size*] [Scheme Procedure]

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`.

7.5.34 SRFI-88 Keyword Objects

SRFI-88 (`http://srfi.schemers.org/srfi-88/srfi-88.html`) provides *keyword objects*, which are equivalent to Guile's keywords (see Section 6.6.8 [Keywords], page 178). SRFI-88 keywords can be entered using the *postfix keyword syntax*, which consists of an identifier followed by `:` (see Section 6.17.2 [Scheme Read], page 346). 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 6.6.8.4 [Keyword Procedures], page 181).

```
(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"
```

7.5.35 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 471.

`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.

7.6 R6RS Support

See Section 6.19.6 [R6RS Libraries], page 372, 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. More generally, it does not make a sharp distinction between binary and textual ports (see Section 6.14.10.6 [R6RS Port Manipulation], page 324).

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 *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 `use-modules` form, say. For example, the R6RS "composite" library can be imported by:

```
(import (rnrs (6)))
(use-modules ((rnrs) :version (6)))
```

For more information on Guile's library implementation, see (see Section 6.19.6 [R6RS Libraries], page 372).

7.6.2.2 rnrs base

The `(rnrs base (6))` library exports the procedures and syntactic forms described in the main section of the Report (see Section "Base library" in *The Revised^6 Report on the Algorithmic Language Scheme*). They are grouped below by the existing manual sections to which they correspond.

`boolean?` *obj* [Scheme Procedure]
`not` *x* [Scheme Procedure]

See Section 6.6.1 [Booleans], page 104, for documentation.

`symbol?` *obj* [Scheme Procedure]
`symbol->string` *sym* [Scheme Procedure]
`string->symbol` *str* [Scheme Procedure]

See Section 6.6.7.4 [Symbol Primitives], page 172, for documentation.

`char?` *obj* [Scheme Procedure]
`char=?` [Scheme Procedure]
`char<?` [Scheme Procedure]
`char>?` [Scheme Procedure]
`char<=?` [Scheme Procedure]
`char>=?` [Scheme Procedure]
`integer->char` *n* [Scheme Procedure]
`char->integer` *chr* [Scheme Procedure]

See Section 6.6.3 [Characters], page 129, for documentation.

`list?` *x* [Scheme Procedure]
`null?` *x* [Scheme Procedure]

See Section 6.7.2.2 [List Predicates], page 185, for documentation.

`pair?` *x* [Scheme Procedure]
`cons` *x y* [Scheme Procedure]
`car` *pair* [Scheme Procedure]
`cdr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cddr` *pair* [Scheme Procedure]
`caaar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]

`caddr` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`caaaar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cdddar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]

See Section 6.7.1 [Pairs], page 182, for documentation.

`number?` *obj* [Scheme Procedure]

See Section 6.6.2.1 [Numerical Tower], page 105, for documentation.

`string?` *obj* [Scheme Procedure]

See Section 6.6.5.2 [String Predicates], page 142, for documentation.

`procedure?` *obj* [Scheme Procedure]

See Section 6.9.7 [Procedure Properties], page 245, for documentation.

`define` *name value* [Scheme Syntax]
`set!` *variable-name value* [Scheme Syntax]

See Section 3.1.3 [Definition], page 16, for documentation.

`define-syntax` *keyword expression* [Scheme Syntax]
`let-syntax` ((*keyword transformer*) . . .) [Scheme Syntax]

exp1 exp2 . . .

`letrec-syntax` ((*keyword transformer*) . . .) [Scheme Syntax]

exp1 exp2 . . . See Section 6.10.1 [Defining Macros], page 248, for documentation.

`identifier-syntax` *exp* [Scheme Syntax]

See Section 6.10.6 [Identifier Macros], page 261, for documentation.

`syntax-rules` *literals* (*pattern template*) ... [Scheme Syntax]

See Section 6.10.2 [Syntax Rules], page 249, for documentation.

`lambda` *formals body* [Scheme Syntax]

See Section 6.9.1 [Lambda], page 236, for documentation.

`let` *bindings body* [Scheme Syntax]
`let*` *bindings body* [Scheme Syntax]
`letrec` *bindings body* [Scheme Syntax]
`letrec*` *bindings body* [Scheme Syntax]

See Section 6.12.2 [Local Bindings], page 277, for documentation.

`let-values` *bindings body* [Scheme Syntax]
`let*-values` *bindings body* [Scheme Syntax]

See Section 7.5.10 [SRFI-11], page 562, for documentation.

`begin` *expr1 expr2 ...* [Scheme Syntax]

See Section 6.13.1 [begin], page 280, for documentation.

`quote` *expr* [Scheme Syntax]
`quasiquote` *expr* [Scheme Syntax]
`unquote` *expr* [Scheme Syntax]
`unquote-splicing` *expr* [Scheme Syntax]

See Section 6.17.1.1 [Expression Syntax], page 343, for documentation.

`if` *test consequence* [*alternate*] [Scheme Syntax]
`cond` *clause1 clause2 ...* [Scheme Syntax]
`case` *key clause1 clause2 ...* [Scheme Syntax]

See Section 6.13.2 [Conditionals], page 281, for documentation.

`and` *expr ...* [Scheme Syntax]
`or` *expr ...* [Scheme Syntax]

See Section 6.13.3 [and or], page 282, for documentation.

`eq?` *x y* [Scheme Procedure]
`eqv?` *x y* [Scheme Procedure]
`equal?` *x y* [Scheme Procedure]
`symbol=?` *symbol1 symbol2 ...* [Scheme Procedure]

See Section 6.11.1 [Equality], page 265, for documentation.

`symbol=?` is identical to `eq?`.

`complex?` *z* [Scheme Procedure]

See Section 6.6.2.4 [Complex Numbers], page 112, 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* [Scheme Procedure]
`even?` *n* [Scheme Procedure]
`gcd` *x ...* [Scheme Procedure]
`lcm` *x ...* [Scheme Procedure]
`exact-integer-sqrt` *k* [Scheme Procedure]

See Section 6.6.2.7 [Integer Operations], page 115, for documentation.

`=` [Scheme Procedure]
`<` [Scheme Procedure]
`>` [Scheme Procedure]
`<=` [Scheme Procedure]
`>=` [Scheme Procedure]
`zero?` *x* [Scheme Procedure]
`positive?` *x* [Scheme Procedure]
`negative?` *x* [Scheme Procedure]

See Section 6.6.2.8 [Comparison], page 117, for documentation.

`for-each` *f lst1 lst2 ...* [Scheme Procedure]

See Section 7.5.3.5 [SRFI-1 Fold and Map], page 538, for documentation.

`list` *elem . . .* [Scheme Procedure]

See Section 6.7.2.3 [List Constructors], page 186, for documentation.

`length` *lst* [Scheme Procedure]
`list-ref` *lst k* [Scheme Procedure]
`list-tail` *lst k* [Scheme Procedure]
See Section 6.7.2.4 [List Selection], page 186, for documentation.

`append` *lst ... obj* [Scheme Procedure]
`append` [Scheme Procedure]
`reverse` *lst* [Scheme Procedure]
See Section 6.7.2.5 [Append/Reverse], page 187, for documentation.

`number->string` *n* [*radix*] [Scheme Procedure]
`string->number` *str* [*radix*] [Scheme Procedure]
See Section 6.6.2.9 [Conversion], page 117, for documentation.

`string` *char ...* [Scheme Procedure]
`make-string` *k* [*chr*] [Scheme Procedure]
`list->string` *lst* [Scheme Procedure]
See Section 6.6.5.3 [String Constructors], page 143, for documentation.

`string->list` *str* [*start* [*end*]] [Scheme Procedure]
See Section 6.6.5.4 [List/String Conversion], page 144, for documentation.

`string-length` *str* [Scheme Procedure]
`string-ref` *str k* [Scheme Procedure]
`string-copy` *str* [*start* [*end*]] [Scheme Procedure]
`substring` *str start* [*end*] [Scheme Procedure]
See Section 6.6.5.5 [String Selection], page 144, 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 147, for documentation.

`string-append` *arg ...* [Scheme Procedure]
See Section 6.6.5.10 [Reversing and Appending Strings], page 154, for documentation.

`string-for-each` *proc s* [*start* [*end*]] [Scheme Procedure]
See Section 6.6.5.11 [Mapping Folding and Unfolding], page 155, for documentation.

`+` *z1 ...* [Scheme Procedure]
`-` *z1 z2 ...* [Scheme Procedure]
`*` *z1 ...* [Scheme Procedure]
`/` *z1 z2 ...* [Scheme Procedure]
`max` *x1 x2 ...* [Scheme Procedure]
`min` *x1 x2 ...* [Scheme Procedure]
`abs` *x* [Scheme Procedure]
`truncate` *x* [Scheme Procedure]
`floor` *x* [Scheme Procedure]

`ceiling` *x* [Scheme Procedure]
`round` *x* [Scheme Procedure]

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 $x = q * y + r$ and $-abs(y/2) <= r < abs(y/2)$. `div0-and-mod0` returns both q and r, and is more efficient than computing each separately.

Note that `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 $-abs(y/2) <= r < abs(y/2)$.

```
(div0 123 10) ⇒ 12
(mod0 123 10) ⇒ 3
(div0-and-mod0 123 10) ⇒ 12 and 3
(div0-and-mod0 123 -10) ⇒ -12 and 3
(div0-and-mod0 -123 10) ⇒ -12 and -3
(div0-and-mod0 -123 -10) ⇒ 12 and -3
(div0-and-mod0 -123.2 -63.5) ⇒ 2.0 and 3.8
(div0-and-mod0 16/3 -10/7) ⇒ -4 and -8/21
```

`real-valued?` *obj* [Scheme Procedure]
`rational-valued?` *obj* [Scheme Procedure]
`integer-valued?` *obj* [Scheme Procedure]

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.

`real-valued?` will return `#t` for complex numbers whose imaginary parts are zero.

`nan?` *x* [Scheme Procedure]
`infinite?` *x* [Scheme Procedure]
`finite?` *x* [Scheme Procedure]

`nan?` returns `#t` if *x* is a NaN value, `#f` otherwise. `infinite?` returns `#t` if *x* is an infinite value, `#f` otherwise. `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.

`assert` *expr* [Scheme Syntax]

Raises an `&assertion` condition if *expr* evaluates to `#f`; otherwise evaluates to the value of *expr*.

`error` *who message irritant1 ...* [Scheme Procedure]
`assertion-violation` *who message irritant1 ...* [Scheme Procedure]

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 ...`.

`error` produces a compound condition with the simple conditions described above, as well as an `&error` condition; `assertion-violation` produces one that includes an `&assertion` condition.

`vector-map` *proc v* [Scheme Procedure]
`vector-for-each` *proc v* [Scheme Procedure]

These procedures implement the `map` and `for-each` contracts over vectors.

`vector` *arg ...* [Scheme Procedure]
`vector?` *obj* [Scheme Procedure]
`make-vector` *len* [Scheme Procedure]
`make-vector` *len fill* [Scheme Procedure]
`list->vector` *l* [Scheme Procedure]
`vector->list` *v* [Scheme Procedure]

See Section 6.7.3.2 [Vector Creation], page 191, for documentation.

`vector-length` *vector* [Scheme Procedure]
`vector-ref` *vector k* [Scheme Procedure]
`vector-set!` *vector k obj* [Scheme Procedure]
`vector-fill!` *v fill* [Scheme Procedure]

See Section 6.7.3.3 [Vector Accessors], page 191, for documentation.

`call-with-current-continuation` *proc* [Scheme Procedure]
`call/cc` *proc* [Scheme Procedure]

See Section 6.13.6 [Continuations], page 288, for documentation.

`values` *arg ...* [Scheme Procedure]
`call-with-values` *producer consumer* [Scheme Procedure]

See Section 6.13.7 [Multiple Values], page 290, for documentation.

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
See Section 6.13.10 [Dynamic Wind], page 298, for documentation.

`apply` *proc arg . . . arglst* [Scheme Procedure]
See Section 6.17.4 [Fly Evaluation], page 348, for documentation.

7.6.2.3 rnrs unicode

The `(rnrs unicode (6))` library provides procedures for manipulating Unicode characters and strings.

`char-upcase` *char* [Scheme Procedure]
`char-downcase` *char* [Scheme Procedure]
`char-titlecase` *char* [Scheme Procedure]
`char-foldcase` *char* [Scheme Procedure]
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.

`char-ci=?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci<?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci>?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci<=?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci>=?` *char1 char2 char3 ...* [Scheme Procedure]
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.

`char-alphabetic?` *char* [Scheme Procedure]
`char-numeric?` *char* [Scheme Procedure]
`char-whitespace?` *char* [Scheme Procedure]
`char-upper-case?` *char* [Scheme Procedure]
`char-lower-case?` *char* [Scheme Procedure]
`char-title-case?` *char* [Scheme Procedure]
These procedures implement various Unicode character set predicates. They are identical to the procedures provided by Guile's core library. See Section 6.6.3 [Characters], page 129, for documentation.

`char-general-category` *char* [Scheme Procedure]
See Section 6.6.3 [Characters], page 129, for documentation.

`string-upcase` *string* [Scheme Procedure]
`string-downcase` *string* [Scheme Procedure]
`string-titlecase` *string* [Scheme Procedure]
`string-foldcase` *string* [Scheme Procedure]
These procedures perform Unicode case folding operations on their input. See Section 6.6.5.9 [Alphabetic Case Mapping], page 153, for documentation.

`string-ci=?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci<?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci>?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci<=?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci>=?` *string1 string2 string3 ...* [Scheme Procedure]

These procedures perform case-insensitive comparison on their input. See Section 6.6.5.7 [String Comparison], page 147, for documentation.

`string-normalize-nfd` *string* [Scheme Procedure]
`string-normalize-nfkd` *string* [Scheme Procedure]
`string-normalize-nfc` *string* [Scheme Procedure]
`string-normalize-nfkc` *string* [Scheme Procedure]

These procedures perform Unicode string normalization operations on their input. See Section 6.6.5.7 [String Comparison], page 147, for documentation.

7.6.2.4 rnrs bytevectors

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 6.6.6 [Bytevectors], page 162.

7.6.2.5 rnrs lists

The `(rnrs lists (6))` library provides procedures additional procedures for working with lists.

`find` *proc list* [Scheme Procedure]

This procedure is identical to the one defined in Guile's SRFI-1 implementation. See Section 7.5.3.7 [SRFI-1 Searching], page 542, for documentation.

`for-all` *proc list1 list2 ...* [Scheme Procedure]
`exists` *proc list1 list2 ...* [Scheme Procedure]

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 542, for documentation.

`filter` *proc list* [Scheme Procedure]
`partition` *proc list* [Scheme Procedure]

These procedures are identical to the ones provided by SRFI-1. See Section 6.7.2.6 [List Modification], page 188, for a description of `filter`; See Section 7.5.3.6 [SRFI-1 Filtering and Partitioning], page 541, for `partition`.

`fold-left` *combine nil list1 list2 . . .* [Scheme Procedure]
`fold-right` *combine nil list1 list2 . . .* [Scheme Procedure]

These procedures are identical to the `fold` and `fold-right` procedures provided by SRFI-1. See Section 7.5.3.5 [SRFI-1 Fold and Map], page 538, for documentation.

`remp` *proc list* [Scheme Procedure]
`remove` *obj list* [Scheme Procedure]
`remv` *obj list* [Scheme Procedure]

`remq` *obj list* [Scheme Procedure]

`remove`, `remv`, and `remq` are identical to the `delete`, `delv`, and `delq` procedures provided by Guile's core library, (see Section 6.7.2.6 [List Modification], page 188). `remp` is identical to the alternate `remove` procedure provided by SRFI-1; See Section 7.5.3.8 [SRFI-1 Deleting], page 544.

`memp` *proc list* [Scheme Procedure]
`member` *obj list* [Scheme Procedure]
`memv` *obj list* [Scheme Procedure]
`memq` *obj list* [Scheme Procedure]

`member`, `memv`, and `memq` are identical to the procedures provided by Guile's core library; See Section 6.7.2.7 [List Searching], page 189, 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* [Scheme Procedure]
`assoc` *obj alist* [Scheme Procedure]
`assv` *obj alist* [Scheme Procedure]
`assq` *obj alist* [Scheme Procedure]

`assoc`, `assv`, and `assq` are identical to the procedures provided by Guile's core library; See Section 6.7.11.1 [Alist Key Equality], page 220, for their documentation. `assp` uses the specified predicate function `proc` to test keys in the association list *alist*.

`cons*` *obj1 ... obj* [Scheme Procedure]
`cons*` *obj* [Scheme Procedure]

This procedure is identical to the one exported by Guile's core library. See Section 6.7.2.3 [List Constructors], page 186, for documentation.

7.6.2.6 rnrs sorting

The `(rnrs sorting (6))` library provides procedures for sorting lists and vectors.

`list-sort` *proc list* [Scheme Procedure]
`vector-sort` *proc vector* [Scheme Procedure]

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 268, for a discussion of the behavior of that procedure.

`vector-sort!` *proc vector* [Scheme Procedure]

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 268, 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 ...* [Scheme Syntax]
`unless` *test expression1 expression2 ...* [Scheme Syntax]

The `when` form is evaluated by evaluating the specified *test* expression; if the result is a true value, the *expression*s that follow it are evaluated in order, and the value of the final *expression* becomes the value of the entire `when` expression.

The `unless` form behaves similarly, with the exception that the specified *expression*s are only evaluated if the value of *test* is false.

`do` ((*variable init step*) ...) (*test expression ...*) *command ...* [Scheme Syntax]

This form is identical to the one provided by Guile's core library. See Section 6.13.4 [while do], page 283, for documentation.

`case-lambda` *clause ...* [Scheme Syntax]

This form is identical to the one provided by Guile's core library. See Section 6.9.5 [Case-lambda], page 243, 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 6.7.8 [Records], page 212, and Section 7.5.8 [SRFI-9], page 558, 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* [Scheme Procedure]

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* [Scheme Procedure]
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* [Scheme Procedure]
Returns the record predicate procedure for the record-type descriptor *rtd*.

`record-accessor` *rtd k* [Scheme Procedure]
Returns the record field accessor procedure for the *k*th field of the record-type descriptor *rtd*.

`record-mutator` *rtd k* [Scheme Procedure]
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 (6))` library provides procedures useful for accessing metadata about R6RS records.

`record?` *obj* [Scheme Procedure]
Return `#t` if the specified object is a non-opaque R6RS record, `#f` otherwise.

`record-rtd` *record* [Scheme Procedure]
Returns the record-type descriptor for *record*. An `&assertion` is raised if *record* is opaque.

`record-type-name` *rtd* [Scheme Procedure]
Returns the name of the record-type descriptor *rtd*.

`record-type-parent` *rtd* [Scheme Procedure]
Returns the parent of the record-type descriptor *rtd*, or `#f` if it has none.

`record-type-uid` *rtd* [Scheme Procedure]
Returns the uid of the record-type descriptor *rtd*, or `#f` if it has none.

`record-type-generative?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is generative, `#f` otherwise.

`record-type-sealed?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is sealed, `#f` otherwise.

`record-type-opaque?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is opaque, `#f` otherwise.

`record-type-field-names` *rtd* [Scheme Procedure]
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* [Scheme Procedure]
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 291, and by the SRFI-18 and SRFI-34 modules—See Section 7.5.15.5 [SRFI-18 Exceptions], page 567, and Section 7.5.22 [SRFI-34], page 578, 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-continuably, another exception, with the condition type `&non-continuable`, will be raised when the exception handler returns locally. Raising an exception continuably 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* [Scheme Procedure]
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 294).

`guard` (*variable clause1 clause2 ...*) *body* [Scheme Syntax]
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 281). 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

```
(guard (ex ((eq? ex 'foo) 'bar) ((eq? ex 'bar) 'baz))
  (raise 'bar))
```

evaluates to `baz`.

`raise` *obj* [Scheme Procedure]

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* [Scheme Procedure]

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 603.

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 7.5.23 [SRFI-35], page 578). 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 predicate field-spec ...* [Scheme Syntax]

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-spec*s, each of which must be of the form:

```
(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` [Condition Type]
`make-message-condition` *message* [Scheme Procedure]
`message-condition?` *obj* [Scheme Procedure]
`condition-message` *condition* [Scheme Procedure]

A type that includes a message describing the condition that occurred.

`&warning` [Condition Type]
`make-warning` [Scheme Procedure]
`warning?` *obj* [Scheme Procedure]

A base type for representing non-fatal conditions during execution.

`&serious` [Condition Type]
`make-serious-condition` [Scheme Procedure]
`serious-condition?` *obj* [Scheme Procedure]

A base type for conditions representing errors serious enough that cannot be ignored.

`&error` [Condition Type]
`make-error` [Scheme Procedure]
`error?` *obj* [Scheme Procedure]

A base type for conditions representing errors.

`&violation` [Condition Type]
`make-violation` [Scheme Procedure]
`violation?` [Scheme Procedure]

A subtype of `&serious` that can be used to represent violations of a language or library standard.

`&assertion` [Condition Type]
`make-assertion-violation` [Scheme Procedure]
`assertion-violation?` *obj* [Scheme Procedure]

A subtype of `&violation` that indicates an invalid call to a procedure.

`&irritants` [Condition Type]
`make-irritants-condition` *irritants* [Scheme Procedure]
`irritants-condition?` *obj* [Scheme Procedure]
`condition-irritants` *condition* [Scheme Procedure]

A base type used for storing information about the causes of another condition in a compound condition.

`&who` [Condition Type]
`make-who-condition` *who* [Scheme Procedure]
`who-condition?` *obj* [Scheme Procedure]
`condition-who` *condition* [Scheme Procedure]

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` [Condition Type]
`make-non-continuable-violation` [Scheme Procedure]
`non-continuable-violation?` *obj* [Scheme Procedure]

A subtype of `&violation` used to indicate that an exception handler invoked by `raise` has returned locally.

`&implementation-restriction` [Condition Type]
`make-implementation-restriction-violation` [Scheme Procedure]
`implementation-restriction-violation?` *obj* [Scheme Procedure]

A subtype of `&violation` used to indicate a violation of an implementation restriction.

`&lexical` [Condition Type]
`make-lexical-violation` [Scheme Procedure]
`lexical-violation?` *obj* [Scheme Procedure]

A subtype of `&violation` used to indicate a syntax violation at the level of the datum syntax.

`&syntax` [Condition Type]
`make-syntax-violation` *form subform* [Scheme Procedure]
`syntax-violation?` *obj* [Scheme Procedure]
`syntax-violation-form` *condition* [Scheme Procedure]
`syntax-violation-subform` *condition* [Scheme Procedure]

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` [Condition Type]
`make-undefined-violation` [Scheme Procedure]
`undefined-violation?` *obj* [Scheme Procedure]

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` [Condition Type]
`make-i/o-error` [Scheme Procedure]
`i/o-error?` *obj* [Scheme Procedure]

A condition supertype for more specific I/O errors.

`&i/o-read` [Condition Type]
`make-i/o-read-error` [Scheme Procedure]
`i/o-read-error?` *obj* [Scheme Procedure]

A subtype of `&i/o`; represents read-related I/O errors.

`&i/o-write` [Condition Type]
`make-i/o-write-error` [Scheme Procedure]
`i/o-write-error?` *obj* [Scheme Procedure]

A subtype of `&i/o`; represents write-related I/O errors.

`&i/o-invalid-position` [Condition Type]
`make-i/o-invalid-position-error` *position* [Scheme Procedure]
`i/o-invalid-position-error?` *obj* [Scheme Procedure]
`i/o-error-position` *condition* [Scheme Procedure]

A subtype of `&i/o`; represents an error related to an attempt to set the file position to an invalid position.

`&i/o-filename` [Condition Type]
`make-io-filename-error` *filename* [Scheme Procedure]
`i/o-filename-error?` *obj* [Scheme Procedure]
`i/o-error-filename` *condition* [Scheme Procedure]

A subtype of `&i/o`; represents an error related to an operation on a named file.

`&i/o-file-protection` [Condition Type]
`make-i/o-file-protection-error` *filename* [Scheme Procedure]
`i/o-file-protection-error?` *obj* [Scheme Procedure]

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` [Condition Type]
`make-i/o-file-is-read-only-error` *filename* [Scheme Procedure]
`i/o-file-is-read-only-error?` *obj* [Scheme Procedure]

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` [Condition Type]
`make-i/o-file-already-exists-error` *filename* [Scheme Procedure]
`i/o-file-already-exists-error?` *obj* [Scheme Procedure]

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` [Condition Type]
`make-i/o-file-does-not-exist-error` [Scheme Procedure]
`i/o-file-does-not-exist-error?` *obj* [Scheme Procedure]

A subtype of `&i/o-filename`; represents an error related to an operation on a non-existent file that was assumed to exist.

`&i/o-port` [Condition Type]
`make-i/o-port-error` *port* [Scheme Procedure]
`i/o-port-error?` *obj* [Scheme Procedure]
`i/o-error-port` *condition* [Scheme Procedure]

A subtype of `&i/o`; represents an error related to an operation on the port *port*.

7.6.2.15 rnrs io ports

The `(rnrs io ports (6))` library provides various procedures and syntactic forms for use in writing to and reading from ports. This functionality is documented in its own section of the manual; (see Section 6.14.10 [R6RS I/O Ports], page 319).

7.6.2.16 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 [I/O Conditions], page 611). 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 6.14.10 [R6RS I/O Ports], page 319, 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 304, 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 6.14.9.1 [File Ports], page 314, for documentation.

`close-input-port` *input-port* [Scheme Procedure]
`close-output-port` *output-port* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.14.4 [Closing], page 309, for documentation.

`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 6.14.2 [Reading], page 306, for documentation.

`read` [Scheme Procedure]
`read` *textual-input-port* [Scheme Procedure]

This procedure is identical to the one provided by Guile's core library. See Section 6.17.2 [Scheme Read], page 346, 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* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.14.3 [Writing], page 308, for documentation.

7.6.2.17 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 460, for documentation.

7.6.2.18 rnrs programs

The `(rnrs programs (6))` library provides procedures for process management and introspection.

`command-line` [Scheme Procedure]

This procedure is identical to the one provided by Guile's core library. See Section 7.2.6 [Runtime Environment], page 471, for documentation.

`exit` [Scheme Procedure]
`exit` *obj* [Scheme Procedure]

This procedure is identical to the one provided by Guile's core library.

7.6.2.19 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* [Scheme Procedure]

Returns `#t` if *obj* is a fixnum, `#f` otherwise.

`fixnum-width` [Scheme Procedure]
`least-fixnum` [Scheme Procedure]
`greatest-fixnum` [Scheme Procedure]

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 ...* [Scheme Procedure]
`fx>?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx<?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx>=?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx<=?` *fx1 fx2 fx3 ...* [Scheme Procedure]

These procedures return `#t` if their fixnum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; `#f` otherwise.

`fxzero?` *fx* [Scheme Procedure]
`fxpositive?` *fx* [Scheme Procedure]
`fxnegative?` *fx* [Scheme Procedure]
`fxodd?` *fx* [Scheme Procedure]
`fxeven?` *fx* [Scheme Procedure]

These numerical predicates return `#t` if *fx* is, respectively, zero, greater than zero, less than zero, odd, or even; `#f` otherwise.

`fxmax` *fx1 fx2 ...* [Scheme Procedure]
`fxmin` *fx1 fx2 ...* [Scheme Procedure]

These procedures return the maximum or minimum of their arguments.

`fx+` *fx1 fx2* [Scheme Procedure]
`fx*` *fx1 fx2* [Scheme Procedure]

These procedures return the sum or product of their arguments.

`fx-` *fx1 fx2* [Scheme Procedure]
`fx-` *fx* [Scheme Procedure]

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.

`fxdiv-and-mod` *fx1 fx2* [Scheme Procedure]
`fxdiv` *fx1 fx2* [Scheme Procedure]
`fxmod` *fx1 fx2* [Scheme Procedure]
`fxdiv0-and-mod0` *fx1 fx2* [Scheme Procedure]
`fxdiv0` *fx1 fx2* [Scheme Procedure]
`fxmod0` *fx1 fx2* [Scheme Procedure]

These procedures implement number-theoretic division on fixnums; See ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩, for a description of their semantics.

`fx+/carry` *fx1 fx2 fx3* [Scheme Procedure]

Returns the two fixnum results of the following computation:

```
(let* ((s (+ fx1 fx2 fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`fx-/carry` *fx1 fx2 fx3* [Scheme Procedure]

Returns the two fixnum results of the following computation:

```
(let* ((d (- fx1 fx2 fx3))
       (d0 (mod0 d (expt 2 (fixnum-width))))
       (d1 (div0 d (expt 2 (fixnum-width)))))
  (values d0 d1))
```

`fx*/carry` *fx1 fx2 fx3* [Scheme Procedure]

```
Returns the two fixnum results of the following computation:
(let* ((s (+ (* fx1 fx2) fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`fxnot` *fx* [Scheme Procedure]
`fxand` *fx1 ...* [Scheme Procedure]
`fxior` *fx1 ...* [Scheme Procedure]
`fxxor` *fx1 ...* [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 124, for documentation.

`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*.

`fxbit-count` *fx* [Scheme Procedure]
Returns the number of 1 bits in the two's complement representation of *fx*.

`fxlength` *fx* [Scheme Procedure]
Returns the number of bits necessary to represent *fx*.

`fxfirst-bit-set` *fx* [Scheme Procedure]
Returns the index of the least significant 1 bit in the two's complement representation of *fx*.

`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.

`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*.

`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).

`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*.

`fxarithmetic-shift` *fx1 fx2* [Scheme Procedure]
`fxarithmetic-shift-left` *fx1 fx2* [Scheme Procedure]
`fxarithmetic-shift-right` *fx1 fx2* [Scheme Procedure]
Returns the result of shifting the bits of *fx1* right or left by the *fx2* positions. `fxarithmetic-shift` is identical to `fxarithmetic-shift-left`.

`fxrotate-bit-field` *fx1 fx2 fx3 fx4* [Scheme Procedure]
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` *fx1 fx2 fx3* [Scheme Procedure]
Returns the result of reversing the order of the bits of *fx1* between position *fx2* (inclusive) and position *fx3* (exclusive).

7.6.2.20 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 Procedure]
Returns `#t` if *obj* is a flonum, `#f` otherwise.

`real->flonum` *x* [Scheme Procedure]
Returns the flonum that is numerically closest to the real number *x*.

`fl=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl<?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl<=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl>?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl>=?` *fl1 fl2 fl3 ...* [Scheme Procedure]

These procedures return `#t` if their flonum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; `#f` otherwise.

`flinteger?` *fl* [Scheme Procedure]
`flzero?` *fl* [Scheme Procedure]
`flpositive?` *fl* [Scheme Procedure]
`flnegative?` *fl* [Scheme Procedure]
`flodd?` *fl* [Scheme Procedure]
`fleven?` *fl* [Scheme Procedure]

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* [Scheme Procedure]
`flinfinite?` *fl* [Scheme Procedure]
`flnan?` *fl* [Scheme Procedure]

These numerical predicates return `#t` if *fl* is, respectively, not infinite, infinite, or a `NaN` value.

`flmax` *fl1 fl2 ...* [Scheme Procedure]
`flmin` *fl1 fl2 ...* [Scheme Procedure]

These procedures return the maximum or minimum of their arguments.

`fl+` *fl1 ...* [Scheme Procedure]
`fl*` *fl ...* [Scheme Procedure]

These procedures return the sum or product of their arguments.

`fl-` *fl1 fl2 ...* [Scheme Procedure]
`fl-` *fl* [Scheme Procedure]
`fl/` *fl1 fl2 ...* [Scheme Procedure]
`fl/` *fl* [Scheme Procedure]

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 *fl*.

`flabs` *fl* [Scheme Procedure]

Returns the absolute value of *fl*.

`fldiv-and-mod` *fl1 fl2* [Scheme Procedure]
`fldiv` *fl1 fl2* [Scheme Procedure]
`fldmod` *fl1 fl2* [Scheme Procedure]
`fldiv0-and-mod0` *fl1 fl2* [Scheme Procedure]
`fldiv0` *fl1 fl2* [Scheme Procedure]

`flmod0` *fl1 fl2* [Scheme Procedure]

These procedures implement number-theoretic division on flonums; See ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩, for a description for their semantics.

`flnumerator` *fl* [Scheme Procedure]
`fldenominator` *fl* [Scheme Procedure]

These procedures return the numerator or denominator of *fl* as a flonum.

`flfloor` *fl1* [Scheme Procedure]
`flceiling` *fl* [Scheme Procedure]
`fltruncate` *fl* [Scheme Procedure]
`flround` *fl* [Scheme Procedure]

These procedures are identical to the `floor`, `ceiling`, `truncate`, and `round` procedures provided by Guile's core library. See Section 6.6.2.11 [Arithmetic], page 119, for documentation.

`flexp` *fl* [Scheme Procedure]
`fllog` *fl* [Scheme Procedure]
`fllog` *fl1 fl2* [Scheme Procedure]
`flsin` *fl* [Scheme Procedure]
`flcos` *fl* [Scheme Procedure]
`fltan` *fl* [Scheme Procedure]
`flasin` *fl* [Scheme Procedure]
`flacos` *fl* [Scheme Procedure]
`flatan` *fl* [Scheme Procedure]
`flatan` *fl1 fl2* [Scheme Procedure]

These procedures, which compute the usual transcendental functions, are the flonum variants of the procedures provided by the R6RS base library (see ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩).

`flsqrt` *fl* [Scheme Procedure]

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* [Scheme Procedure]

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?` [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 `NaN`s.

`fixnum->flonum` *fx* [Scheme Procedure]

Returns the flonum that is numerically closest to the fixnum *fx*.

7.6.2.21 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 7.5.30 [SRFI-60], page 586).

`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 124, 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 `i`th bit from *ei2* if the `i`th bit of *ei1* is 1, the `i`th 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* [Scheme Procedure]

Returns the index of the least significant 1 bit in the two's complement representation of *ei*.

`bitwise-bit-set?` *ei1 ei2* [Scheme Procedure]

Returns `#t` if the *ei2*th bit in the two's complement representation of *ei1* is 1, `#f` otherwise.

`bitwise-copy-bit` *ei1 ei2 ei3* [Scheme Procedure]

Returns the result of setting the *ei2*th bit of *ei1* to the *ei2*th bit of *ei3*.

`bitwise-bit-field` *ei1 ei2 ei3* [Scheme Procedure]

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-copy-bit-field` *ei1 ei2 ei3 ei4* [Scheme Procedure]

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* [Scheme Procedure]
`bitwise-arithmetic-shift-left` *ei1 ei2* [Scheme Procedure]
`bitwise-arithmetic-shift-right` *ei1 ei2* [Scheme Procedure]

Returns the result of shifting the bits of *ei1* right or left by the *ei2* positions. `bitwise-arithmetic-shift` is identical to `bitwise-arithmetic-shift-left`.

`bitwise-rotate-bit-field` *ei1 ei2 ei3 ei4* [Scheme Procedure]

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* [Scheme Procedure]

Returns the result of reversing the order of the bits of *ei1* between position *ei2* (inclusive) and position *ei3* (exclusive).

7.6.2.22 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 253, for documentation, examples, and rationale.

`make-variable-transformer` *proc* [Scheme Procedure]

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* ... [Scheme Syntax]

The `syntax-case` pattern matching form.

`syntax` *template* [Scheme Syntax]
`quasisyntax` *template* [Scheme Syntax]
`unsyntax` *template* [Scheme Syntax]
`unsyntax-splicing` *template* [Scheme Syntax]

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 253, for documentation.

`identifier?` *obj* [Scheme Procedure]
`bound-identifier=?` *id1 id2* [Scheme Procedure]
`free-identifier=?` *id1 id2* [Scheme Procedure]

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* [Scheme Procedure]

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* [Scheme Procedure]
`datum->syntax` *template-id datum* [Scheme Procedure]

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* [Scheme Procedure]
`syntax-violation` *whom message form subform* [Scheme Procedure]

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

7.6.2.23 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 6.7.13 [Hash Tables], page 228, and Section 7.5.33 [SRFI-69], page 588, 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` [Scheme Procedure]
`make-eq-hashtable` *k* [Scheme Procedure]

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` [Scheme Procedure]
`make-eqv-hashtable` *k* [Scheme Procedure]

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* [Scheme Procedure]
`make-hashtable` *hash-function equiv k* [Scheme Procedure]

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* [Scheme Procedure]
Returns `#t` if *obj* is an R6RS hash table, `#f` otherwise.

`hashtable-size` *hashtable* [Scheme Procedure]
Returns the number of keys currently in the hash table *hashtable*.

`hashtable-ref` *hashtable key default* [Scheme Procedure]
Returns the value associated with *key* in the hash table *hashtable*, or *default* if none is found.

`hashtable-set!` *hashtable key obj* [Scheme Procedure]
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* [Scheme Procedure]
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* [Scheme Procedure]
Returns `#t` if the hash table *hashtable* contains an association for the key *key*, `#f` otherwise.

`hashtable-update!` *hashtable key proc default* [Scheme Procedure]
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* [Scheme Procedure]
`hashtable-copy` *hashtable mutable* [Scheme Procedure]
Returns a copy of the hash table *hashtable*. If the optional argument *mutable* is a true value, the new hash table will be immutable.

`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 6.7.13.2 [Hash Table Reference], page 230, 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.

7.6.2.24 rnrs enums

The `(rnrs enums (6))` library provides structures and procedures for working with enumerable 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 operations on lists; Guile's `(rnrs enums)` implementation makes use of several of them. See Section 7.5.3.10 [SRFI-1 Set Operations], page 545, 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* [Scheme Procedure]
`enum-set-subset?` *enum-set1 enum-set2* [Scheme Procedure]
`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* [Scheme Procedure]
`enum-set-intersection` *enum-set1 enum-set2* [Scheme Procedure]
`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 ...*) *constructor-syntax* [Scheme 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-set*s in the universe specified by `(symbol ...)`; and a "predicate macro" bound to *type-name*, which has the following form:

```
(type-name sym)
```

If *sym* is a member of the universe specified by the *symbol*s above, this form evaluates to *sym*. Otherwise, a `&syntax` condition is raised.

7.6.2.25 rnrs

The `(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:

- `(rnrs eval (6))`
- `(rnrs mutable-pairs (6))`
- `(rnrs mutable-strings (6))`
- `(rnrs r5rs (6))`

7.6.2.26 rnrs eval

The `(rnrs eval (6)` library provides procedures for performing "on-the-fly" evaluation of expressions.

`eval` *expression environment* [Scheme Procedure]

Evaluates *expression*, which must be a datum representation of a valid Scheme expression, in the environment specified by *environment*. This procedure is identical to the

one provided by Guile's code library; See Section 6.17.4 [Fly Evaluation], page 348, for documentation.

`environment` *import-spec ...* [Scheme Procedure]

Constructs and returns a new environment based on the specified *import-spec*s, which must be datum representations of the import specifications used with the `import` form. See Section 6.19.6 [R6RS Libraries], page 372, for documentation.

7.6.2.27 rnrs mutable-pairs

The `(rnrs mutable-pairs (6))` library provides the `set-car!` and `set-cdr!` procedures, which allow the `car` and `cdr` fields of a pair to be modified.

These procedures are identical to the ones provide by Guile's core library. See Section 6.7.1 [Pairs], page 182, for documentation. All pairs in Guile are mutable; consequently, these procedures will never throw the `&assertion` condition described in the R6RS libraries specification.

7.6.2.28 rnrs mutable-strings

The `(rnrs mutable-strings (6))` library provides the `string-set!` and `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 146, for documentation. All strings in Guile are mutable; consequently, these procedures will never throw the `&assertion` condition described in the R6RS libraries specification.

7.6.2.29 rnrs r5rs

The `(rnrs r5rs (6))` library exports bindings for some procedures present in R5RS but omitted from the R6RS base library specification.

`exact->inexact` *z* [Scheme Procedure]
`inexact->exact` *z* [Scheme Procedure]

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* [Scheme Procedure]
`remainder` *n1 n2* [Scheme Procedure]
`modulo` *n1 n2* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.6.2.7 [Integer Operations], page 115, for documentation.

`delay` *expr* [Scheme Syntax]
`force` *promise* [Scheme Procedure]

The `delay` form and the `force` procedure are identical to their counterparts in Guile's core library. See Section 6.17.9 [Delayed Evaluation], page 356, for documentation.

`null-environment` *n* [Scheme Procedure]
`scheme-report-environment` *n* [Scheme Procedure]

These procedures are identical to the ones provided by the `(ice-9 r5rs)` Guile module. See Section 6.19.12 [Environments], page 381, 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            ;; <- the input object
    (('hello (who))   ;; <- the pattern
     who)))           ;; <- 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`.

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*.

The pattern matcher is defined as follows:

`match` *exp clause1 clause2* ... [Scheme Syntax]

Match object *exp* against the patterns in *clause1 clause2* ... 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 `(pattern body1 body2 ...)`. Each *pattern* must follow the syntax described below. Each body is an arbitrary Scheme expression, possibly referring to pattern variables of *pattern*.

The syntax and interpretation of patterns is as follows:

```
        patterns:                       matches:

pat ::= identifier                      anything, and binds identifier
      | _                               anything
      | ()                              the empty list
      | #t                              #t
      | #f                              #f
```

```
      | string                          a string
      | number                          a number
      | character                       a character
      | 'sexp                           an s-expression
      | 'symbol                         a symbol (special case of s-expr)
      | (pat_1 ... pat_n)               list of n elements
      | (pat_1 ... pat_n . pat_{n+1})   list of n or more
      | (pat_1 ... pat_n pat_n+1 ooo)   list of n or more, each element
                                          of remainder must match pat_n+1
      | #(pat_1 ... pat_n)              vector of n elements
      | #(pat_1 ... pat_n pat_n+1 ooo)  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"`.

Please refer to the `ice-9/match.upstream.scm` file in your Guile installation for more details.

Guile also comes with a pattern matcher specifically tailored to SXML trees, See Section 7.16 [sxml-match], page 655.

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 6.22.3 [Runtime Options], page 416).

`readline-options` [Scheme Procedure]
`readline-enable` *option-name* [Scheme Procedure]
`readline-disable` *option-name* [Scheme Procedure]
`readline-set!` *option-name value* [Scheme Syntax]

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-parens   500     Time (ms) to show matching opening parenthesis
                        (0 = off).
```

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 6.14.9.1 [File Ports], page 314), and should usually be a terminal.

The default is the `current-input-port` and `current-output-port` (see Section 6.14.8 [Default Ports], page 313) 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 652) 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 482) then enable readline for all reading from `current-input-port` (see Section 6.14.8 [Default Ports], page 313) 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 652) 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 466), 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 `define`s). 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* [Function]

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*] [Scheme Procedure]

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 6.14.3 [Writing], page 308)

`#: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.

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[2], 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?` *flag*
: If *flag* is true then print using `display`. The default is `#f` which means use `write` style. (see Section 6.14.3 [Writing], page 308)

`#:width` *columns*
: Print within the given *columns*. The default is 79.

`#:breadth-first?` *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.

C programmers will note the similarity between `format` and `printf`, though escape sequences are marked with `~` instead of `%`, and are more powerful.

`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 6.14.8 [Default Ports], page 313), or `#f` to return the output as a string.

[2] On Unicode-capable ports, the ellipsis is represented by character 'HORIZONTAL ELLIPSIS' (U+2026), otherwise it is represented by three dots.

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 6.14.3 [Writing], page 308).

```
(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 *minwidth*. The default *minpad* is 0 and the default *padinc* is 1 (imposing no minimum or multiple).

```
(format #f "~5,1,4a" 'abc) ⇒ "abc    "
```

`~c`
: Character. Parameter: *charnum*.

  Output a character. The default is to simply output, as per `write-char` (see Section 6.14.3 [Writing], page 308). `~@c` prints in `write` style. `~:c` prints control characters (ASCII 0 to 31) in `^X` form.

```
(format #t "~c" #\z)         ⊣ z
(format #t "~@c" #\z)        ⊣ #\z
(format #t "~:c" #\newline) ⊣ ^J
```

If the *charnum* parameter is given then an argument is not taken but instead the character is `(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.

```
(format #t "~65c")  ⊣ A
```

`~d`
`~x`
`~o`
`~b` Integer. Parameters: *minwidth*, *padchar*, *commachar*, *commawidth*.

Output an integer argument as a decimal, hexadecimal, octal or binary integer (respectively), in a locale-independent way.

```
(format #t "~d" 123) ⊣ 123
```

`~@d` etc shows a `+` sign is shown on positive numbers.

```
(format #t "~@b" 12) ⊣ +1100
```

If the output is less than the *minwidth* parameter (default no minimum), it's padded on the left with the *padchar* parameter (default space).

```
(format #t "~5,'*d" 12)   ⊣ ***12
(format #t "~5,'0d" 12)   ⊣ 00012
(format #t "~3d"    1234) ⊣ 1234
```

`~:d` adds commas (or the *commachar* parameter) every three digits (or the *commawidth* parameter many). However, when your intent is to write numbers in a way that follows typographical conventions, using `~h` is recommended.

```
(format #t "~:d" 1234567)         ⊣ 1,234,567
(format #t "~10,'*,'/,2:d" 12345) ⊣ ***1/23/45
```

Hexadecimal `~x` output is in lower case, but the `~(` and `~)` case conversion directives described below can be used to get upper case.

```
(format #t "~x"       65261) ⊣ feed
(format #t "~:@(~x~)" 65261) ⊣ FEED
```

`~r` Integer in words, roman numerals, or a specified radix. Parameters: *radix*, *minwidth*, *padchar*, *commachar*, *commawidth*.

With no parameters output is in words as a cardinal like "ten", or `~:r` prints an ordinal like "tenth".

```
(format #t "~r" 9)  ⊣ nine        ;; cardinal
(format #t "~r" -9) ⊣ minus nine  ;; cardinal
(format #t "~:r" 9) ⊣ ninth       ;; ordinal
```

And also with no parameters, `~@r` gives roman numerals and `~:@r` gives old roman numerals. In old roman numerals there's no "subtraction", so 9 is `VIIII` instead of `IX`. In both cases only positive numbers can be output.

```
(format #t "~@r" 89)   ⊣ LXXXIX     ;; roman
(format #t "~:@r" 89)  ⊣ LXXXVIIII  ;; old roman
```

When a parameter is given it means numeric output in the specified *radix*. The modifiers and parameters following the radix are the same as described for `~d` etc above.

```
(format #f "~3r" 27)   ⇒ "1000"    ;; base 3
(format #f "~3,5r" 26) ⇒ "  222"   ;; base 3 width 5
```

`~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.

```
(format #t "~f" 5)       ⊣ 5.0
(format #t "~f" "123")   ⊣ 123.0
(format #t "~f" "1e-1")  ⊣ 0.1
```

`~@f` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@f" 0)  ⊣ +0.0
```

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.

```
(format #f "~6f" -1.5)      ⇒ "  -1.5"
(format #f "~6,,,,'*f" 23)  ⇒ "**23.0"
(format #f "~6f" 1234567.0) ⇒ "1234567.0"
```

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.)

```
(format #t "~1,2f" 3.125) ⊣ 3.13
(format #t "~1,2f" 1.5)   ⊣ 1.50
```

scale is a power of 10 applied to the value, moving the decimal point that many places. A positive *scale* increases the value shown, a negative decreases it.

```
(format #t "~,,2f" 1234)  ⊣ 123400.0
(format #t "~,,-2f" 1234) ⊣ 12.34
```

If *overflowchar* and *width* are both given and if the output would exceed *width*, then that many *overflowchar*s are printed instead of the value.

```
(format #t "~6,,,'xf" 12345) ⊣ 12345.
(format #t "~5,,,'xf" 12345) ⊣ xxxxx
```

`~h` Localized number[3]. 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 6.24.4 [Number Input and Output], page 423).

[3] The `~h` format specifier first appeared in Guile version 2.0.6.

```
(format #t "~h" 12345.5678)  ; with "C" as the current locale
⊣ 12345.5678

(format #t "~14,,'*:h" 12345.5678
        (make-locale LC_ALL "en_US"))
⊣ ***12,345.5678

(format #t "~,2:h" 12345.5678
        (make-locale LC_NUMERIC "fr_FR"))
⊣ 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) ⊣ 5.00025E+3
(format #t "~e" "123.4") ⊣ 1.234E+2
(format #t "~e" "1e4")   ⊣ 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) ⊣ +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)     ⇒ "  1.234E+3"
(format #f "~10,,,,,'*e" 0.5) ⇒ "****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) ⇒ "1.111E+4"
(format #f "~,8e" 123.0)   ⇒ "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) ⇒ "1.0E+99"
(format #f "~,,6e" 1.0e99) ⇒ "1.0E+000099"
```

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 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 *overflowchar*s 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 "~,,,,,,'ee" 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 "~$" 5)       ⊣ 5.00
(format #t "~4$" "2.25") ⊣ 2.2500
(format #t "~4$" "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 6.24 [Internationalization], page 420).

`~y` Structured printing. Parameters: *width*.

`~y` outputs an argument using `pretty-print` (see Section 7.9 [Pretty Printing], page 632). 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 632). 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.

`~?`
`~k` 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.

~*

Argument jumping. Parameter: *N*.

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 ~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 "~#*~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.

~t

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 6.14.2 [Reading], page 306), 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).

```
(format #f "~v_foo" 4) ⇒ "    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 `force-output` to flush any buffers on the destination (see Section 6.14.3 [Writing], page 308). `~!` 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.

```
(format #f "abc~
            ~d def~
            ~d" 1 2) ⇒ "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,

```
(format #t "~(Hello~)")    ⊣ hello
(format #t "~:@(Hello~)") ⊣ 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 outermost 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.

```
(format #t "~{~d~}"     '(1 2 3))        ⊣ 123
(format #t "~{~s=~d ~}" '("x" 1 "y" 2)) ⊣ "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.

```
(format #t "~:{~dx~d ~}" '((1 2) (3 4) (5 6)))
⊣ 1x2 3x4 5x6
```

`~@{` takes arguments directly, with each iteration successively consuming arguments.

```
(format #t "~@{~d~}"     1 2 3)        ⊣ 123
(format #t "~@{~s=~d ~}" "x" 1 "y" 2) ⊣ "x"=1 "y"=2
```

`~:@{` takes list arguments, one argument for each iteration, using that list for the format.

```
(format #t "~:@{~dx~d ~}" '(1 2) '(3 4) '(5 6))
⊣ 1x2 3x4 5x6
```

Iterating stops when there are no more arguments or when the *maxreps* parameter to `~{` is reached (default no maximum).

```
(format #t "~2{~d~}" '(1 2 3 4)) ⊣ 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.

```
(format #t "~{~}" "~d" '(1 2 3)) ⊣ 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.

```
(format #t "~{~{~d~}x~}" '((1 2) (3 4 5))) ⊣ 12x345x
```

See also `~^` below for escaping from iteration.

`~[ ~; ~]` Conditional. Parameter: *selector*.

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~^/~} 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 *val1* $\leq$ *val2* and *val2* $\leq$ *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 conditionalize the endpoint of an iteration.

```
(format #t "~{ ~[ ... ~] ~}" ...)       ;; good
(format #t "~{ ~[ ... ~} ... ~]" ...)   ;; bad
```

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 6.14.3 [Writing], page 308), 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:

```
(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-glil.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 460).

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)))

  ;; Count zero bytes for directories.
  (define (down name stat result) result)
  (define (up name stat result) result)

  ;; Likewise for skipped directories.
  (define (skip name stat result) result)

  ;; Ignore unreadable files/directories but warn the user.
  (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 6.24.2 [Text Collation], page 422). 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 460) 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 6.7.13.2 [Hash Table Reference], page 230)

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*] [*'physical*] [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 460). *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 6.7.13.2 [Hash Table Reference], page 230)

`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` [Scheme Procedure]
: Return a new queue.

`q?` *obj* [Scheme Procedure]
: 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* [Scheme Procedure]
: Add *obj* to the rear of *q*, and return *q*.

`deq!` *q* [Scheme Procedure]
`q-pop!` *q* [Scheme Procedure]
: 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* [Scheme Procedure]
: Add *obj* to the front of *q*, and return *q*.

`q-length` *q* [Scheme Procedure]
: Return the number of elements in *q*.

`q-empty?` *q* [Scheme Procedure]

Return true if *q* is empty.

`q-empty-check` *q* [Scheme Procedure]

Throw a `q-empty` exception if *q* is empty.

`q-front` *q* [Scheme Procedure]

Return the first element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-rear` *q* [Scheme Procedure]

Return the last element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-remove!` *q obj* [Scheme Procedure]

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* [Scheme Procedure]

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

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 6.17.9 [Delayed Evaluation], page 356), 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 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 [Reading], page 306) 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 290), 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 290) 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 6.17.2 [Scheme Read], page 346),

```
(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* ... [Macro]

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)))
("^da(em)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 336. 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 336. 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:

```
(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* . . . [Macro]

`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:

```
(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 8.4 [sxml simple], page 669, for more information on SXML.

The following example[4] provides a brief illustration, transforming a music album catalog language into HTML.

[4] 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.

```
(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 626), `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 `sxml-match` may include a *guard expression*.

The pattern notation is based on that of Scheme's `syntax-rules` and `syntax-case` macro systems. The grammar for the `sxml-match` syntax is given below:

```
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*])
                  | (unquote [cata-expression -> var-symbol*])
```

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 `sxml-match` pattern matcher.

Matching XML Elements

The example below illustrates the pattern matching of an XML element:

```
(sxml-match '(e (@ (i 1)) 3 4 5)
  [(e (@ (i ,d)) ,a ,b ,c) (list d a b c)]
  [,otherwise #f])
```

Each clause in `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 `e` with an attribute `i` and three children.

Pattern variables are must be "unquoted" in the pattern. The above expression binds *d* to `1`, *a* to `3`, *b* to `4`, and *c* to `5`.

Ellipses in Patterns

As in `syntax-rules`, ellipses may be used to specify a repeated pattern. Note that the pattern `item ...` specifies zero-or-more matches of the pattern `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 `a` element, within an element `d`.

```
(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 ...) ...)])
```

The above expression returns a value of `((1 2 3) (4 5) (6 7 8) (9 10))`.

Ellipses in Quasiquote'd Output

Within the body of an `sxml-match` form, a slightly extended version of quasiquote is provided, which allows the use of ellipses. This is illustrated in the example below.

```
(sxml-match '(e 3 4 5 6 7)
  [(e ,i ... 6 7) `("start" ,(list 'wrap i) ... "end")]
  [,otherwise #f])
```

The general pattern is that `` `(something ,i ...) `` is rewritten as `` `(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.

```
(sxml-match '("i" "j" "k" "l" "m")
  [(list ,a ,b ,c ,d ,e)
   `((p ,a) (p ,b) (p ,c) (p ,d) (p ,e))])
```

This example wraps each nodeset item in an HTML paragraph element. This example can be rewritten and simplified through using ellipsis:

```
(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:

```
(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:

```
(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:

```
(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:

```
(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:

```
(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`.

```
(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 `,[x]` recursively invokes the pattern matcher on the value bound in this position.

```
(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.

```
(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 ...))
       `(div (p ,start-time
                (br) ,series-title
                (br) ,desc ...))]
      [(Program (Start ,start-time) (Duration ,dur) (Series ,series-title)
                (Description ,desc ...)
                ,[cast-list -> cl])
       `(div (p ,start-time
                (br) ,series-title
                (br) ,desc ...)
             ,cl)]))
  (sxml-match g
    [(TVGuide (@ (start ,start-date)
                 (end ,end-date))
              (Channel (Name ,nm) ,[prog -> p] ...) ...)
     `(html (head (title "TV Guide"))
            (body (h1 "TV Guide")
```

```
                    (div (h2 ,nm) ,p ...) ...))]))
```

sxml-match-let and sxml-match-let*

sxml-match-let ((*pat expr*) ...) *expression0 expression ...* [Scheme Syntax]
sxml-match-let* ((*pat expr*) ...) *expression0 expression ...* [Scheme Syntax]

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.

8 Standard Library

8.1 (statprof)

8.1.1 Overview

`(statprof)` is intended to be a fairly simple statistical profiler for guile. It is in the early stages yet, so consider its output still suspect, and please report any bugs to `guile-devel at gnu.org`, or to me directly at `rlb at defaultvalue.org`.

A simple use of statprof would look like this:

```
(statprof-reset 0 50000 #t)
(statprof-start)
(do-something)
(statprof-stop)
(statprof-display)
```

This would reset statprof, clearing all accumulated statistics, then start profiling, run some code, stop profiling, and finally display a gprof flat-style table of statistics which will look something like this:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 35.29     0.23     0.23     2002     0.11     0.11  -
 23.53     0.15     0.15     2001     0.08     0.08  positive?
 23.53     0.15     0.15     2000     0.08     0.08  +
 11.76     0.23     0.08     2000     0.04     0.11  do-nothing
  5.88     0.64     0.04     2001     0.02     0.32  loop
  0.00     0.15     0.00        1     0.00   150.59  do-something
...
```

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.

% time — The percent of the time spent inside the procedure itself (not counting children).

cumulative seconds — The total number of seconds spent in the procedure, including children.

self seconds — The total number of seconds spent in the procedure itself (not counting children).

calls — The total number of times the procedure was called.

self ms/call — The average time taken by the procedure itself on each call, in ms.

total ms/call — The average time taken by each call to the procedure, including time spent in child functions.

name The name of the procedure.

The profiler uses `eq?` and the procedure object itself to identify the procedures, so it won't confuse different procedures with the same name. They will show up as two different rows in the output.

Right now the profiler is quite simplistic. I cannot provide call-graphs or other higher level information. What you see in the table is pretty much all there is. Patches are welcome :-)

8.1.2 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 is run which looks at the current procedure that's executing, and then crawls up the stack, and for each procedure encountered, increments that procedure's sample count. Note that if a procedure is encountered multiple times on a given stack, it is only counted once. After the sampling is complete, the profiler resets profiling timer to fire again after the appropriate interval.

Meanwhile, the profiler keeps track, via `get-internal-run-time`, how much CPU time (system and user – which is also what `ITIMER_PROF` tracks), has elapsed while code has been executing within a statprof-start/stop block.

The profiler also tries to avoid counting or timing its own code as much as possible.

8.1.3 Usage

`statprof-active?` [Function]

Returns `#t` if `statprof-start` has been called more times than `statprof-stop`, `#f` otherwise.

`statprof-start` [Function]

Start the profiler.

`statprof-stop` [Function]

Stop the profiler.

`statprof-reset` *sample-seconds sample-microseconds count-calls?* [*full-stacks?*] [Function]

Reset the statprof sampler 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 *full-stacks?* is true, collect all sampled stacks into a list for later analysis.

Enables traps and debugging as necessary.

`statprof-accumulated-time` [Function]

Returns the time accumulated during the last statprof run.

`statprof-sample-count` [Function]

Returns the number of samples taken during the last statprof run.

`statprof-fold-call-data` *proc init* [Function]

Fold *proc* over the call-data accumulated by statprof. Cannot be called while statprof is active. *proc* should take two arguments, `(call-data prior-result)`.

Note that a given proc-name may appear multiple times, but if it does, it represents different functions with the same name.

`statprof-proc-call-data` *proc* [Function]

Returns the call-data associated with *proc*, or `#f` if none is available.

`statprof-call-data-name` *cd* [Function]

`statprof-call-data-calls` *cd* [Function]

`statprof-call-data-cum-samples` *cd* [Function]

`statprof-call-data-self-samples` *cd* [Function]

`statprof-call-data->stats` *call-data* [Function]

Returns an object of type `statprof-stats`.

`statprof-stats-proc-name` *stats* [Function]

`statprof-stats-%-time-in-proc` *stats* [Function]

`statprof-stats-cum-secs-in-proc` *stats* [Function]

`statprof-stats-self-secs-in-proc` *stats* [Function]

`statprof-stats-calls` *stats* [Function]

`statprof-stats-self-secs-per-call` *stats* [Function]

`statprof-stats-cum-secs-per-call` *stats* [Function]

`statprof-display` . _ [Function]

Displays a gprof-like summary of the statistics collected. Unless an optional *port* argument is passed, uses the current output port.

`statprof-display-anomolies` [Function]

A sanity check that attempts to detect anomolies in statprof's statistics.

`statprof-fetch-stacks` [Function]

Returns a list of stacks, as they were captured since the last call to `statprof-reset`.

Note that stacks are only collected if the *full-stacks?* argument to `statprof-reset` is true.

`statprof-fetch-call-tree` [Function]

```
Return a call tree for the previous statprof run.

The return value is a list of nodes, each of which is of the type:
@@code
 node ::= (@@var@{proc@} @@var@{count@} . @@var@{nodes@})
@@end code
```

`statprof` *thunk* [*#:loop*] [*#:hz*] [*#:count-calls?*] [*#:full-stacks?*] [Function]

Profiles the execution of *thunk*.

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.

If *full-stacks?* is true, at each sample, statprof will store away the whole call tree, for later analysis. Use `statprof-fetch-stacks` or `statprof-fetch-call-tree` to retrieve the last-stored stacks.

`with-statprof` *args* [Special Form]

Profiles the expressions in its body.

Keyword arguments:

`#:loop` Execute the body *loop* number of times, or `#f` for no looping
default: `#f`

`#:hz` Sampling rate
default: `20`

`#:count-calls?`
Whether to instrument each function call (expensive)
default: `#f`

`#:full-stacks?`
Whether to collect away all sampled stacks into a list
default: `#f`

`gcprof` *thunk* [*#:loop*] [*#:full-stacks?*] [Function]

Do an allocation profile of the execution of *thunk*.

The stack will be sampled soon after every garbage collection, 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.

If *full-stacks?* is true, at each sample, statprof will store away the whole call tree, for later analysis. Use `statprof-fetch-stacks` or `statprof-fetch-call-tree` to retrieve the last-stored stacks.

8.2 (sxml apply-templates)

8.2.1 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.

8.2.2 Usage

`apply-templates` *tree templates* [Function]

8.3 (sxml fold)

8.3.1 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`.

8.3.2 Usage

`foldt` *fup fhere tree* [Function]

The standard multithreaded tree fold.

fup is of type [a] -> a. *fhere* is of type object -> a.

`foldts` *fdown fup fhere seed tree* [Function]

The single-threaded tree fold originally defined in SSAX. See Section 8.5 [(sxml ssax)], page 669, for more information.

`foldts*` *fdown fup fhere seed tree* [Function]

A variant of [foldts], page 667 that allows pre-order tree rewrites. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation.*

`fold-values` *proc list . seeds* [Function]

A variant of Section 7.5.3.5 [SRFI-1 Fold and Map], page 538 that allows multi-valued seeds. Note that the order of the arguments differs from that of `fold`.

`foldts*-values` *fdown fup fhere tree . seeds* [Function]

A variant of [foldts*], page 667 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* [Function]

A traversal combinator in the spirit of SSAX's [pre-post-order], page 674.

`fold-layout` was originally presented in Andy Wingo's 2007 paper, *Applications of fold to XML transformation.*

```
bindings := (<binding>...)
binding  := (<tag> <bandler-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.

8.4 (sxml simple)

8.4.1 Overview

A simple interface to XML parsing and serialization.

8.4.2 Usage

`xml->sxml` [*port*] [Function]

Use SSAX to parse an XML document into SXML. Takes one optional argument, *port*, which defaults to the current input port.

`sxml->xml` *tree* [*port*] [Function]

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* [Function]

Detag an sxml tree *sxml* into a string. Does not perform any formatting.

8.5 (sxml ssax)

8.5.1 Overview

Functional XML parsing framework

SAX/DOM and SXML parsers with support for XML Namespaces and validation

This 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, this file includes a semi-validating SXML parser.

The present XML framework 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 callbacks 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 functions 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 his 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

```
parser:: (Start-tag -> Seed -> Seed) ->
  (Start-tag -> Seed -> Seed -> Seed) ->
  (Char-Data -> Seed -> Seed) ->
  XML-text-fragment -> Seed -> Seed
parser fdown fup fchar "<elem attrs> content </elem>" seed
 = fup "<elem attrs>" seed
(parser fdown fup fchar "content" (fdown "<elem attrs>" seed))

parser fdown fup fchar "char-data content" seed
 = parser fdown fup fchar "content" (fchar "char-data" seed)

parser fdown fup fchar "elem-content content" seed
 = parser fdown fup fchar "content" (
parser fdown fup fchar "elem-content" seed)
```

Compare the last two equations with the left fold

```
fold-left kons elem:list seed = 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:

[1] Jeremy Gibbons, Geraint Jones, "The Under-appreciated Unfold," Proc. ICFP'98, 1998, pp. 273-279.

[2] Richard S. Bird, The promotion and accumulation strategies in transformational programming, ACM Trans. Progr. Lang. Systems, 6(4):487-504, October 1984.

[3] Ralf Hinze, "Deriving Backtracking Monad Transformers," Functional Pearl. Proc ICFP'00, pp. 186-197.

8.5.2 Usage

`current-ssax-error-port` [Function]

`with-ssax-error-to-port` *port thunk* [Function]

`xml-token? _` [Function]

```
-- Scheme Procedure: pair? x
    Return ‘#t’ if X is a pair; otherwise return ‘#f’.
```

`xml-token-kind` *token* [Special Form]

`xml-token-head` *token* [Special Form]

`make-empty-attlist` [Function]

`attlist-add` *attlist name-value* [Function]

`attlist-null? _` [Function]

```
-- Scheme Procedure: null? x
    Return ‘#t’ iff X is the empty list, else ‘#f’.
```

`attlist-remove-top` *attlist* [Function]

`attlist->alist` *attlist* [Function]

`attlist-fold` *kons knil lis1* [Function]

`define-parsed-entity!` *entity str* [Function]

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!` [Function]

Restore the set of parsed entity definitions to its initial state.

`ssax:uri-string->symbol` *uri-str* [Function]

`ssax:skip-internal-dtd` *port* [Function]

`ssax:read-pi-body-as-string` *port* [Function]

`ssax:reverse-collect-str-drop-ws` *fragments* [Function]

`ssax:read-markup-token` *port* [Function]

`ssax:read-cdata-body` *port str-handler seed* [Function]

`ssax:read-char-ref` *port* [Function]

`ssax:read-attributes` *port entities* [Function]

`ssax:complete-start-tag` *tag-head port elems entities namespaces* [Function]

`ssax:read-external-id` *port* [Function]

`ssax:read-char-data` *port expect-eof? str-handler seed* [Function]

`ssax:xml->sxml` *port namespace-prefix-assig* [Function]

`ssax:make-parser` . *kw-val-pairs* [Special Form]

`ssax:make-pi-parser` *orig-handlers* [Special Form]

`ssax:make-elem-parser` *my-new-level-seed my-finish-element my-char-data-handler my-pi-handlers* [Special Form]

8.6 (sxml ssax input-parse)

8.6.1 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.

8.6.2 Usage

`peek-next-char` [*port*] [Function]

`assert-curr-char` *expected-chars comment* [*port*] [Function]

`skip-until` *arg* [*port*] [Function]

`skip-while` *skip-chars* [*port*] [Function]

`next-token` *prefix-skipped-chars break-chars* [*comment*] [*port*] [Function]

`next-token-of` *incl-list/pred* [*port*] [Function]

`read-text-line` [*port*] [Function]

`read-string` *n* [*port*] [Function]

`find-string-from-port? _ _ . _` [Function]

Looks for *str* in <*input-port*>, optionally within the first *max-no-char* characters.

8.7 (sxml transform)

8.7.1 Overview

SXML expression tree transformers

Pre-Post-order traversal of a tree and creation of a new tree

```
pre-post-order:: <tree> x <bindings> -> <new-tree>
```

where

```
 <bindings> ::= (<binding> ...)
 <binding> ::= (<trigger-symbol> *preorder* . <handler>) |
               (<trigger-symbol> *macro* . <handler>) |
(<trigger-symbol> <new-bindings> . <handler>) |
(<trigger-symbol> . <handler>)
 <trigger-symbol> ::= XMLname | *text* | *default*
 <handler> :: <trigger-symbol> x [<tree>] -> <new-tree>
```

The pre-post-order function visits the nodes and nodelists pre-post-order (depth-first). For each `<Node>` of the form `(`*name* `<Node> ...)`, it looks up an association with the given *name* among its <*bindings*>. If failed, `pre-post-order` tries to locate a `*default*` binding. It's an error if the latter attempt fails as well. Having found a binding, the `pre-post-order` function first checks to see if the binding is of the form

```
(<trigger-symbol> *preorder* . <handler>)
```

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

```
(<trigger-symbol> *macro* . <handler>)
```

This is equivalent to `*preorder*` described above. However, the result is re-processed again, with the current stylesheet.

8.7.2 Usage

`SRV:send-reply` . *fragments* [Function]

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 fhere seed tree* [Function]

`post-order` *tree bindings* [Function]

`pre-post-order` *tree bindings* [Function]

`replace-range` *beg-pred end-pred forest* [Function]

8.8 (sxml xpath)

8.8.1 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.scm) is rather suitable for querying as it is. Bowing to the XPath specification, we will refer to SXML information items as 'Nodes':

```
<Node> ::= <Element> | <attributes-coll> | <attrib>
   | "text string" | <PI>
```

This production can also be described as

```
<Node> ::= (name . <Nodeset>) | "text string"
```

An (ordered) set of nodes is just a list of the constituent nodes:

```
<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 singe 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 a 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 case of SXPath the corresponding rules are given as comments to a sxpath function, below. The regression test suite at the end of this file 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, while the others are adjusted for our running example, tree1.

8.8.2 Usage

`nodeset?` *x* [Function]

`node-typeof?` *crit* [Function]

`node-eq?` *other* [Function]

`node-equal?` *other* [Function]

`node-pos` *n* [Function]

`filter` *pred?* [Function]

```
      -- Scheme Procedure: filter pred list
          Return all the elements of 2nd arg LIST that satisfy predicate
          PRED.  The list is not disordered - elements that appear in the
          result list occur in the same order as they occur in the argument
          list. The returned list may share a common tail with the argument
          list. The dynamic order in which the various applications of pred
          are made is not specified.

               (filter even? '(0 7 8 8 43 -4)) => (0 8 8 -4)
```

`take-until` *pred?* [Function]

`take-after` *pred?* [Function]

`map-union` *proc lst* [Function]

`node-reverse` *node-or-nodeset* [Function]

`node-trace` *title* [Function]

`select-kids` *test-pred?* [Function]

`node-self` *pred?* [Function]

```
-- Scheme Procedure: filter pred list
    Return all the elements of 2nd arg LIST that satisfy predicate
    PRED.  The list is not disordered - elements that appear in the
    result list occur in the same order as they occur in the argument
    list. The returned list may share a common tail with the argument
    list. The dynamic order in which the various applications of pred
    are made is not specified.

         (filter even? '(0 7 8 8 43 -4)) => (0 8 8 -4)
```

`node-join` . *selectors* [Function]

`node-reduce` . *converters* [Function]

`node-or` . *converters* [Function]

`node-closure` *test-pred?* [Function]

`node-parent` *rootnode* [Function]

`sxpath` *path* [Function]

8.9 (texinfo)

8.9.1 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'.

8.9.2 Usage

`call-with-file-and-dir` *filename proc* [Function]

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.

`texi-command-specs` [Variable]

`texi-command-depth` *command max-depth* [Function]

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

8.10 (texinfo docbook)

8.10.1 Overview

This module exports procedures for transforming a limited subset of the SXML representation 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.

8.10.2 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 `@chapter`, 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 678, and [filter-empty-elements], page 678.

Returns a nodeset, as described in Section 8.8 [sxml xpath], page 674. That is to say, this function returns an untagged list of stexi elements.

`filter-empty-elements` *sdocbook* [Function]

Filters out empty elements in an sdocbook nodeset. Mostly useful after running `sdocbook-flatten`.

`replace-titles` *sdocbook-fragment* [Function]

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:

```
(replace-titles '((refsect1) (title "Foo") (para "Bar.")))
   ⇒ '((chapter "Foo") (para "Bar."))
```

8.11 (texinfo html)

8.11.1 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 [sxml simple sxml->xml], page 669. References (i.e., the `@ref` family of commands) are resolved by a *ref-resolver*. See [texinfo html add-ref-resolver!], page 679, for more information.

8.11.2 Usage

`add-ref-resolver!` *proc* [Function]

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* [Function]

Transform the stexi *tree* into shtml, resolving references via ref-resolvers. See the module commentary for more details.

`urlify` *str* [Function]

8.12 (texinfo indexing)

8.12.1 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 ;)

8.12.2 Usage

`stexi-extract-index` *tree manual-name kind* [Function]

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).

8.13 (texinfo string-utils)

8.13.1 Overview

Module '`(texinfo string-utils)`' provides various string-related functions useful to Guile's texinfo support.

8.13.2 Usage

`escape-special-chars` *str special-chars escape-char* [Function]

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.

```
;; make a string regexp-safe...
 (escape-special-chars "***(Example String)***"
```

```
                          "[]()/*."
                          #\\)
=> "\\*\\*\\*\\(Example String\\)\\*\\*\\*"

;; also can escape a singe char...
 (escape-special-chars "richardt@vzavenue.net"
                          #\@
                          #\@)
=> "richardt@@vzavenue.net"
```

`transform-string` *str match? replace* [*start*] [*end*] [Function]

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 te 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 *start* and *end* are given, they control which portion of the string undergoes transformation. The entire input string is still output, though. So, if *start* is '`5`', then the first five characters of *str* will still appear in the returned string.

```
; these two are equivalent...
 (transform-string str #\space #\-) ; change all spaces to -'s
 (transform-string str (lambda (c) (char=? #\space c)) #\-)
```

`expand-tabs` *str* [*tab-size*] [Function]

Returns a copy of *str* with all tabs expanded to spaces. *tab-size* defaults to 8.

Assuming tab size of 8, this is equivalent to:

```
 (transform-string str #\tab "        ")
```

`center-string` *str* [*width*] [*chr*] [*rchr*] [Function]

Returns a copy of *str* centered in a field of *width* characters. Any needed padding is done by character *chr*, which defaults to '`#\space`'. If *rchr* is provided, then the padding to the right will use it instead. See the examples below. left and *rchr* on the right. The default *width* is 80. The default *chr* and *rchr* is '`#\space`'. The string is never truncated.

```
(center-string "Richard Todd" 24)
=> "      Richard Todd      "

(center-string " Richard Todd " 24 #\=)
=> "===== Richard Todd =====" 

(center-string " Richard Todd " 24 #\< #\>)
=> "<<<<< Richard Todd >>>>>"
```

`left-justify-string` *str* [*width*] [*chr*] [Function]

`left-justify-string str [width chr]`. Returns a copy of *str* padded with *chr* such that it is left justified in a field of *width* characters. The default *width* is 80. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`right-justify-string` *str* [*width*] [*chr*] [Function]

Returns a copy of *str* padded with *chr* such that it is right justified in a field of *width* characters. The default *width* is 80. The default *chr* is '`#\space`'. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`collapse-repeated-chars` *str* [*chr*] [*num*] [Function]

Returns a copy of *str* with all repeated instances of *chr* collapsed down to at most *num* instances. The default value for *chr* is '`#\space`', and the default value for *num* is 1.

```
(collapse-repeated-chars "H  e  l  l  o")
=> "H e l l o"
(collapse-repeated-chars "H--e--l--l--o" #\-)
=> "H-e-l-l-o"
(collapse-repeated-chars "H-e--l---l----o" #\- 2)
=> "H-e--l--l--o"
```

`make-text-wrapper` [*#:line-width*] [*#:expand-tabs?*] [*#:tab-width*] [*#:collapse-whitespace?*] [*#:subsequent-indent*] [*#:initial-indent*] [*#:break-long-words?*] [Function]

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* [Function]

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* [Function]

`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`.

8.14 (texinfo plain-text)

8.14.1 Overview

Transformation from stexi to plain-text. Strives to re-create the output from `info`; comes pretty damn close.

8.14.2 Usage

`stexi->plain-text` *tree* [Function]

Transform *tree* into plain text. Returns a string.

8.15 (texinfo serialize)

8.15.1 Overview

Serialization of `stexi` to plain texinfo.

8.15.2 Usage

`stexi->texi` *tree* [Function]

Serialize the stexi *tree* into plain texinfo.

8.16 (texinfo reflection)

8.16.1 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.

8.16.2 Usage

`module-stexi-documentation` *sym-name* [*%docs-resolver*] [*#:docs-resolver*] [Function]

Return documentation for the module named *sym-name*. The documentation will be formatted as `stexi` (see Section 8.9 [texinfo], page 676).

`script-stexi-documentation` *scriptpath* [Function]

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 8.9 [texinfo], page 676).

`object-stexi-documentation` _ [_] [*#:force*] [Function]

`package-stexi-standard-copying` *name version updated years copyright-holder permissions* [Function]

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 684, [texinfo reflection package-stexi-standard-titlepage], page 683, [texinfo reflection package-stexi-standard-copying], page 683, and [texinfo reflection package-stexi-standard-menu], page 684.

`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 684, 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*] [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. 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.

9 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))
```

9.1 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.

9.2 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 *superclass*es specifies which existing classes, if any, to inherit slots and properties from. *Slots* hold per-instance[1] 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 *super*s, with direct slots defined by *slot-definition*s and *class-option*s. The newly created class is bound to the variable name *name* in the current environment.

Each *slot-definition* is either a symbol that names the slot or a list,

[1] Usually — but see also the `#:allocation` slot option.

```
(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:

```
(define-class <my-complex> (<number>)
  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* [class option]

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 9.11.1 [Metaobjects and the Metaobject Protocol], page 709 and Section 9.11.2 [Metaclasses], page 711.

If the `#:metaclass` option is absent, GOOPS reuses or constructs a metaclass for the new class by calling `ensure-metaclass` (see Section 9.11.5 [ensure-metaclass], page 713).

`#:name` *name* [class option]

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.

9.3 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

```
(define c (make <my-complex>))
```

creates a new `<my-complex>` object and binds it to the Scheme variable `c`.

`make` [generic]
`make` (*class* `<`*class*`>`) *initarg* . . . [method]

Create and return a new instance of class *class*, initialized using *initarg* . . . .

[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>`.

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 9.4 [init-keyword], page 687). 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` [generic]
`make-instance` (*class* <*class*>) *initarg* . . . [method]

`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
```

9.4 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.

```
(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:

```
(define-class <chbouib> ()
  (hashtab #:init-thunk make-hash-table))
```

or:

```
(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

```
(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 *obj* with slots named `foo` and `bar`, it is always possible to read and write those slots by calling `slot-ref` and `slot-set!` with the relevant slot name; for example:

```
(slot-ref obj 'foo)
(slot-set! obj 'bar 25)
```

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. *getter* specifies a generic function to which GOOPS will add a method for getting the slot value. *setter* specifies a generic function to which GOOPS will add a method for setting the slot value. *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:

```
(c #:getter get-count #:setter set-count #:accessor count)
```

GOOPS defines generic function methods such that the slot value can be referenced using either the getter or the accessor -

```
(let ((current-count (get-count obj))) ...)
(let ((current-count (count obj))) ...)
```

- and set using either the setter or the accessor -

```
(set-count obj (+ 1 current-count))
(set! (count obj) (+ 1 current-count))
```

Note that

- with an accessor, the slot value is set using the generalized `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 `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 `define-class` expression is evaluated. For more detail, see Section 9.11.9 [ensure-generic], page 719.

`#: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 *instance*s 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 definition 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 9.11.6 [Customizing Class Definition], page 716.

`#: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.

getter 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*.

9.5 Illustrating Slot Description

To illustrate slot description, we can redefine the `<my-complex>` class seen before. A definition 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))
(get-r c2) ⇒ 2
(get-i c2) ⇒ 0
```

Accessors can both read and write a slot. So, another definition of the `<my-complex>` class, using the `#:accessor` option, could be:

```
(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:

```
(real-part c)
```

and set with:

```
(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:

```
(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)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a))))))
   (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)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a)))))))
```

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

```
(slot-set! c 'a 3)
```

permits to set the angle of the `c` complex number.

```
(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`.

```
(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)))
```

9.6 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.

```
(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

```
(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:

```
(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* [syntax]

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* ... [syntax]

Define a method for the generic function or accessor *generic* with parameters *parameters* and body *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.

9.6.1 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 246). 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* [syntax]

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.

9.6.2 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* [primitive procedure]

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* [primitive procedure]

Return the generic function definition of *primitive*.

`primitive-generic-generic` raises an error if *primitive* is not a primitive with generic capability.

9.6.3 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 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 6.19.3 [duplicate binding handlers], page 367 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.

9.6.4 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))
```

With these definitions,

```
(Test 1)   ⇒ (integer number top)
(Test 1.0) ⇒ (number 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.

```
(Test #t)  ⇒ (top)
```

`next-method` is always called as just `(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 `next-method`, but instead simply write the new call as usual:

```
(define-method (Test (a <number>) min max)
  (if (and (>= a min) (<= a max))
      (display "Number is in range\n"))
  (Test a))

(Test 2 1 10)
⊣
Number is in range
⇒
(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.)

9.6.5 Generic Function and Method Examples

Consider the following definitions:

```
(define-generic G)
(define-method (G (a <integer>) b) 'integer)
(define-method (G (a <real>) b) 'real)
(define-method (G a b) 'top)
```

The `define-generic` call defines *G* as a generic function. The three next lines define methods for *G*. Each method uses a sequence of *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 `<top>`. Thus, the first method definition is equivalent to

```
(define-method (G (a <integer>) (b <top>)) 'integer)
```

Now, let's look at some possible calls to the generic function *G*:

```
(G 2 3)    ⇒ integer
(G 2 #t)   ⇒ integer
(G 1.2 'a) ⇒ real
(G #t #f)  ⇒ top
(G 1 2 3)  ⇒ error (since no method exists for 3 parameters)
```

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:

```
(define-method (G (a <integer>) (b <number>))  'integer-number)
(define-method (G (a <integer>) (b <real>))    'integer-real)
(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 9.1.

```
(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 9.1: 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.

9.6.6 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.

`no-method` [generic]
`no-method` (*gf* <*generic*>) *args* [method]

When an application invokes a generic function, and no methods at all have been defined for that generic function, GOOPS calls the `no-method` generic function. The default method calls `goops-error` with an appropriate message.

`no-applicable-method` [generic]
`no-applicable-method` (*gf* <*generic*>) *args* [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 `no-applicable-method` generic function. The default method calls `goops-error` with an appropriate message.

`no-next-method` [generic]
`no-next-method` (*gf* <*generic*>) *args* [method]

When a generic function method calls `(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 `no-next-method` generic function. The default method calls `goops-error` with an appropriate message.

9.7 Inheritance

Here are some class definitions to help illustrate inheritance:

```
(define-class A () a)
(define-class B () b)
(define-class C () c)
(define-class D (A B) d a)
(define-class E (A C) e c)
(define-class F (D E) f)
```

`A`, `B`, `C` have a null list of superclasses. In this case, the system will replace the null list by a list which only contains `<object>`, the root of all the classes defined by `define-class`. `D`, `E`, `F` use multiple inheritance: each class inherits from two previously defined classes. Those class definitions define a hierarchy which is shown in Figure 9.2. In this figure, the class `<top>` is also shown; this class is the superclass of all Scheme objects. In particular, `<top>` is the superclass of all standard Scheme types.

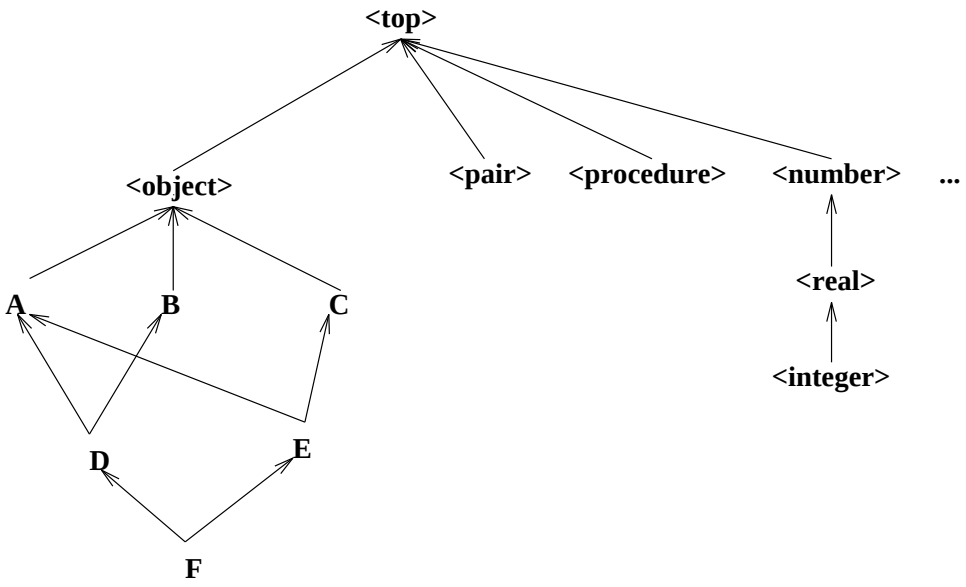

Figure 9.2: A class hierarchy.

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.

9.7.1 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 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.

[6] This section is an adaptation of material from Jeff Dalton's (J.Dalton@ed.ac.uk) *Brief introduction to CLOS*

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

```
(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 9.2, 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:

```
(class-precedence-list B) ⇒ (#<<class> B 401b97c8>
                                        #<<class> <object> 401e4a10>
                                        #<<class> <top> 4026a9d8>)
```

Or for a more immediately readable result:

```
(map class-name (class-precedence-list B)) ⇒ (B <object> <top>)
```

9.7.2 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.

9.8 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.

9.8.1 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 9.7.1 [Class Precedence List], page 700). 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.

9.8.2 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>
```

9.8.3 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* [procedure]
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 9.4 [allocation], page 687), 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 9.4 [getter], page 687), 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 9.4 [setter], page 687), 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 9.4 [accessor], page 687), 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 9.4 [init-value], page 687), 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 9.4 [init-form], page 687), 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 9.4 [init-thunk], page 687), 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 9.4 [init-keyword], page 687), 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 9.4 [init-value], page 687.

9.8.4 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` *gf* [primitive procedure]

Return the name of generic function *gf*.

`generic-function-methods` *gf* [primitive procedure]

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` *method* [primitive procedure]

Return the generic function that *method* belongs to. This is the value of *method*'s `generic-function` slot.

`method-specializers` *method* [primitive procedure]

Return a list of *method*'s formal parameter specializers . This is the value of *method*'s `specializers` slot.

`method-procedure` *method* [primitive procedure]

Return the procedure that implements *method*. This is the value of *method*'s `procedure` slot.

`method-source` [generic]
`method-source` (*m* `<`*method*`>`) [method]

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)))
```

9.8.5 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* [primitive procedure]
Return `#t` if *obj* has a slot with name *slot-name*, otherwise `#f`.

`slot-bound?` *obj slot-name* [primitive procedure]
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 9.8.5 [Accessing Slots], page 706).

`slot-ref` *obj slot-name* [primitive procedure]
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 9.8.5 [Accessing Slots], page 706).

`slot-ref` calls the generic function `slot-unbound` if the named slot in *obj* does not have a value (see Section 9.8.5 [Accessing Slots], page 706).

`slot-set!` *obj slot-name value* [primitive procedure]
Set the value of the slot named *slot-name* in *obj* to *value*.

`slot-set!` calls the generic function `slot-missing` if *obj* does not have a slot called *slot-name* (see Section 9.8.5 [Accessing Slots], page 706).

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* [primitive procedure]
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* [primitive procedure]
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 9.8.5 [Accessing Slots], page 706).

`slot-ref-using-class` *class obj slot-name* [primitive procedure]
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 9.8.5 [Accessing Slots], page 706).

`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 9.8.5 [Accessing Slots], page 706).

`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 9.8.5 [Accessing Slots], page 706).

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 *class*. The named slot must have `#:class` or `#:each-subclass` allocation (see Section 9.4 [allocation], page 687).

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 *class* to *value*. The named slot must have `#:class` or `#:each-subclass` allocation (see Section 9.4 [allocation], page 687).

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.

9.9 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 9.8.5 [slot-missing], page 706)
- `slot-unbound` (see Section 9.8.5 [slot-unbound], page 706)
- `no-method` (see Section 9.6.6 [no-method], page 699)
- `no-applicable-method` (see Section 9.6.6 [no-applicable-method], page 699)
- `no-next-method` (see Section 9.6.6 [no-next-method], page 699)

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`.

9.10 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` [generic]
`shallow-clone` (*self* <*object*>) [method]

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.

`deep-clone` [generic]
`deep-clone` (*self* <*object*>) [method]

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

`write` *object port* [primitive generic]
`display` *object port* [primitive generic]

When GOOPS is loaded, `write` and `display` become generic functions with special methods for printing

- objects - instances of the class `<object>`
- foreign objects - instances of the class `<foreign-object>`
- classes - instances of the class `<class>`
- generic functions - instances of the class `<generic>`
- methods - instances of the class `<method>`.

`write` and `display` print non-GOOPS values in the same way as the Guile primitive `write` and `display` functions.

In addition to the cases mentioned, you can of course define `write` and `display` methods for your own classes, to customize how instances of those classes are printed.

9.11 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 `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.

9.11.1 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 customization of that area.

9.11.2 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> #:supers (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>`. (`#:supers` 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> #:supers (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 9.11.3 [MOP Specification], page 712.

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 `my-object` is `<my-class2>`.
- The class of `<my-class2>` is `<my-metaclass>`.
- The class of `<my-metaclass>` is `<class>`.

9.11.3 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 `(initialize` *`instance initargs`*`)` invocation in the default `make-instance` method is customizable, because the type of the *`instance`* argument is determined by the class that was passed to `make-instance`.

(Whereas — to give a counter-example — the `(make <generic> #:name ',name)` invocation in `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 *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.

9.11.4 Instance Creation Protocol

`make <class> .` *`initargs`* (method)

- `allocate-instance` *`class initargs`* (generic)

  The applied `allocate-instance` method should allocate storage for a new instance of class *class* and return the uninitialized instance.

- `initialize` *`instance initargs`* (generic)

  instance is the uninitialized instance returned by `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.

`make` itself is a generic function. Hence the `make` invocation itself can be customized in the case where the new instance's metaclass is more specialized than the default `<class>`, by defining a `make` method that is specialized to that metaclass.

Normally, however, the method for classes with metaclass `<class>` will be applied. This method calls two generic functions:

- (allocate-instance *class* . *initargs*)
- (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 9.11.5 [Class Definition Protocol], page 713.
- 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.

9.11.5 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)

    - `compute-setter-method` (generic)
- `class-redefinition` (generic)
    - `remove-class-accessors` (generic)
    - `update-direct-method!` (generic)
    - `update-direct-subclass!` (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 9.2 [Class Definition], page 685) expands to an expression which

- checks that it is being evaluated only at top level
- defines any accessors that are implied by the *slot-definition*s
- 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 9.12 [Redefining a Class], page 721).

`class` *name* (*super* ...) *slot-definition* ... *class-option* ... [syntax]

Return a newly created class that inherits from *super*s, with direct slots defined by *slot-definition*s and *class-option*s. For the format of *slot-definition*s and *class-option*s, see Section 9.2 [define-class], page 685.

`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-option*s. For the format of *slots* and *class-option*s, see Section 9.2 [define-class], page 685, 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: `(list get set)`.

The closures returned by `compute-get-n-set` are stored as part of the value of the *class* metaobject's `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

`(slot-name-symbol init-function . index)`

or like

`(slot-name-symbol init-function get set)`

Where the get and set closures are replaced by *index*, the slot is an instance slot and *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, *get* and *set* are the closures returned by `compute-get-n-set`.

The structure of the `getters-n-setters` slot value is important when understanding the next customizable generic functions that `initialize` calls...

- `compute-getter-method` *class gns* (generic)

  `initialize` calls `compute-getter-method` for each of the class's slots (as determined by `compute-slots`) that includes a `#:getter` or `#:accessor` slot option. *gns* is the element of the *class* metaobject's `getters-n-setters` slot that specifies how the slot in question is referenced and set, as described above under `compute-get-n-set`. The applied method should create and return a method that is specialized for instances of type *class* and uses the get closure to retrieve the slot's value. `initialize` uses `add-method!` to add the returned method to the generic function named by the slot definition's `#:getter` or `#:accessor` option.

- `compute-setter-method` *class gns* (generic)

  `compute-setter-method` is invoked with the same arguments as `compute-getter-method`, for each of the class's slots that includes a `#:setter` or `#:accessor` slot option. The applied method should create and return a method that is specialized for instances of type *class* and uses the set closure to set the slot's value. `initialize` then uses `add-method!` to add the returned method to the generic function named by the slot definition's `#:setter` or `#:accessor` option.

9.11.6 Customizing Class Definition

If the metaclass of the new class is something more specialized than the default `<class>`, then the type of *class* in the calls above is more specialized than `<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 `initialize`, `compute-cpl` or `compute-get-n-set`.

`compute-cpl` computes the class precedence list ("CPL") for the new class (see Section 9.7.1 [Class Precedence List], page 700), 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 `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.

`compute-std-cpl` *class* [procedure]

Compute and return the class precedence list for *class* according to the algorithm described in Section 9.7.1 [Class Precedence List], page 700.

`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 9.4 [allocation], page 687). 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.

```
(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 the batch allocation count.
       (set! batch-allocation-count (+ batch-allocation-count 1))
       batch-get-n-set)
```

```
        ;; Call next-method to handle standard allocation types.
        (else (next-method)))))

(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 9.11.5 [Class Definition Protocol], page 713.

`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.

9.11.7 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 9.6.2 [Extending Primitives], page 694)

By defining further methods for `add-method!`, you can theoretically handle adding methods to further types of target.

9.11.8 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 *parameter*s 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 *parameter*s 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 9.6 [define-method], page 692).

`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* [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]

Generic function for adding method *method* to *target*.

`add-method!` (*generic* <*generic*>) (*method* <*method*>) [method]

Add method *method* to the generic function *generic*.

`add-method!` (*proc* <*procedure*>) (*method* <*method*>) [method]

If *proc* is a procedure with generic capability (see Section 9.6.2 [generic-capability?], page 694), upgrade it to a primitive generic and add *method* to its generic function definition.

`add-method!` (*pg* <*primitive-generic*>) (*method* <*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*>) [method]

Raise an error indicating that *whatever* is not a valid generic function.

9.11.9 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*] [procedure]

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.

9.11.10 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.

9.12 Redefining a Class

Suppose that a class `<my-class>` is defined using `define-class` (see Section 9.2 [define-class], page 685), with slots that have accessor functions, and that an application has created several instances of `<my-class>` using `make` (see Section 9.3 [make], page 686). What then happens if `<my-class>` is redefined by calling `define-class` again?

9.12.1 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 9.3 [make], page 686).
- 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. . .

9.12.2 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` [generic]

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>`) [method]

Implements GOOPS' default class redefinition behaviour, as described in Section 9.12.1 [Default Class Redefinition Behaviour], page 721. 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* [generic]

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* [generic]

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* [generic]

The default `update-direct-subclass!` method invokes `class-redefinition` recursively to handle the redefinition of subclasses.

An alternative class redefinition strategy could be to leave all existing instances as instances 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!

9.13 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 `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 9.8.1 [slot-init-function], page 702).

`change-class` *instance new-class* [generic]

`change-class` (*obj* <*object*>) (*new* <*class*>) [method]

Modify instance *obj* to make it an instance of class *new*.

The value of each of *obj*'s slots is preserved only if a similarly named slot exists in *new*; any other slot values are discarded.

The slots in *new* that do not correspond to any of *obj*'s pre-existing slots are initialized according to *new*'s slot definitions' init functions.

The default `change-class` method also invokes another generic function, `update-instance-for-different-class`, as the last thing that it does before returning. The applied `update-instance-for-different-class` method can make any further adjustments to *new-instance* that are required to complete or modify the change of class. The return value from the applied method is ignored.

`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 `update-instance-for-different-class` method does nothing.

Customized change of class behaviour can be implemented by defining `change-class` methods that are specialized either by the class of the instances to be modified or by the metaclass of the new class.

10 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.

10.1 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.

10.1.1 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.

10.1.2 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.

10.1.3 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.

10.1.4 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.

1.3.2 — 20 August 1999
1.3.4 — 25 September 1999
1.4 — 21 June 2000
: 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.

1.6 — 6 September 2002
: 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.

1.8 — 20 February 2006
: 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 orthogonalized in a massive way.

2.0 — 16 February 2010
: 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.

10.1.5 Status, or: Your Help Needed

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.

The addition of the virtual machine in Guile 2.0, together with the compiler infrastructure, should go a long way to addressing the speed issues. But there is much optimization to be done. Interested contributors will find lots of delightful low-hanging fruit, from simple profile-driven optimization to hacking a just-in-time compiler from VM bytecode to native code.

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. Though there is a somewhat-working Emacs Lisp language frontend for Guile, it cannot yet execute all of Emacs Lisp. A serious integration of Guile with Emacs would replace the Elisp virtual machine with Guile, and provide the necessary C shims so that Guile could emulate Emacs' C API. This would 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 implementations. Guile does not yet support the latest Scheme standard, R6RS, and should do so. Like all standards, R6RS 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.

10.2 Data Representation

Scheme is a latently-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 implementations 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.

10.2.1 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:

```
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.

10.2.2 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:

```
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;
};

#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:

  ```
  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:

```
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?)

10.2.3 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 CAR and 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:

```
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:

  ```
  POINTER_P (x) && 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.

10.2.4 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.hpl.hp.com/personal/Hans_Boehm/gc`, for more information.

10.2.5 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`.

10.2.5.1 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 10.2.3 [Cheaper Pairs], page 732). 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.

10.2.5.2 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:

`int SCM_IMP` (*SCM x*) [Macro]

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` [Macro]

The Scheme empty list object, or "End Of List" object, usually written in Scheme as `'()`.

`SCM SCM_EOF_VAL` [Macro]

The Scheme end-of-file value. It has no standard written representation, for obvious reasons.

`SCM SCM_UNSPECIFIED` [Macro]

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` [Macro]

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.

`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.

10.2.5.3 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.

`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!

10.2.5.4 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`.

`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*) [Function]

Like `scm_cell`, but allocates a double cell with four slots.

10.2.5.5 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`) [Macro]

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`) [Macro]

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.

10.2.5.6 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`) [Macro]

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`) [Macro]

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`) [Macro]

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) ⇒ SCM_SET_CELL_WORD (x, 0, w)
- SCM_SET_CELL_WORD_1 (x, w) ⇒ SCM_SET_CELL_WORD (x, 1, w)
- ...
- SCM_SET_CELL_WORD_n (x, w) ⇒ SCM_SET_CELL_WORD (x, n, w)

`void SCM_SET_CELL_OBJECT` (*SCM* `x`, *unsigned int* `n`, *SCM* `o`) [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) ⇒ SCM_SET_CELL_OBJECT (x, 0, o)
- SCM_SET_CELL_OBJECT_1 (x, o) ⇒ SCM_SET_CELL_OBJECT (x, 1, o)
- ...
- SCM_SET_CELL_OBJECT_n (x, o) ⇒ 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.

10.3 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.

10.3.1 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 performs 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. 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 2.0 is slower than the interpreter in 1.8. We hope the that the compiler's speed makes up for the loss!

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 10.4.8 [Extending the Compiler], page 770.

10.3.2 VM Concepts

Compiled code is run by a virtual machine (VM). Each thread has its own VM. When a compiled procedure is run, Guile looks up the virtual machine for the current thread and executes the procedure using that VM.

Guile's virtual machine is a stack machine—that is, it has few registers, and the instructions defined in the VM operate by pushing and popping values from a stack.

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.

A VM has generic instructions, such as those to reference local variables, and instructions designed to support Guile's languages – mathematical instructions that support the entire numerical tower, an inlined implementation of `cons`, etc.

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 stack machines; their exact meanings in the context of Guile's VM are described in the next section.

10.3.3 Stack Layout

While not strictly necessary to understand how to work with the VM, it is instructive and sometimes entertaining to consider the structure of the VM stack.

Logically speaking, a VM stack is composed of "frames". Each frame corresponds to the application of one compiled procedure, and contains storage space for arguments, local variables, intermediate values, and some bookkeeping information (such as what to do after the frame computes its value).

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.)

Within a frame, you have the data associated with the function application itself, which is of a fixed size, and the stack space for intermediate values. Sometimes only the former is referred to as the "frame", and the latter is the "stack", although all pending application frames can have some intermediate computations interleaved on the stack.

The structure of the fixed part of an application frame is as follows:

```
            Stack
| ...               |
| Intermed. val. 0 | <- fp + bp->nargs + bp->nlocs = SCM_FRAME_UPPER_ADDRESS (fp
+==================+
| Local variable 1 |
| Local variable 0 | <- fp + bp->nargs
| Argument 1       |
| Argument 0       | <- fp
| Program          | <- fp - 1
+------------------+
| Return address   |
| MV return address|
| Dynamic link     | <- fp - 4 = SCM_FRAME_DATA_ADDRESS (fp) = SCM_FRAME_LOWER_A
+==================+
|                  |
```

In the above drawing, the stack grows upward. The intermediate values stored in the application of this frame are stored above `SCM_FRAME_UPPER_ADDRESS (fp)`. `bp` refers to the `struct scm_objcode` data associated with the program at `fp - 1`. `nargs` and `nlocs` are properties of the compiled procedure, which will be discussed later.

The individual fields of the frame are as follows:

Return address
: 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`.

MV return address
: The `ip` to return to if this application returns multiple values. For continuations that only accept one value, this value will be `NULL`; for others, it will be an `ip` that points to a multiple-value return address in the calling code. That code will expect the top value on the stack to be an integer—the number of values being returned—and that below that integer there are the values being returned.

Dynamic link
: This is the `fp` in effect before this program was applied. In effect, this and the return address are the registers that are always "saved". The dynamic link links the current frame to the previous frame; computing a stack trace involves traversing these frames.

Local variable *n*
: Lambda-local variables that are all allocated as part of the frame. This makes access to variables very cheap.

Argument *n*
: The calling convention of the VM requires arguments of a function application to be pushed on the stack, and here they are. References to arguments dispatch to these locations on the stack.

Program
: This is the program being applied. For more information on how programs are implemented, See Section 10.3.5 [VM Programs], page 742.

10.3.4 Variables and the VM

Consider the following Scheme code as an example:

```
(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 allows one to prove statements about functions. 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 6.19.7 [Variables], page 373, 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.

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.

10.3.5 Compiled Procedures are VM Programs

By default, when you enter in expressions at Guile's REPL, they are first compiled to VM object code, then that VM object code 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, a reference to any captured lexical variables, an object array, and some metadata such as the procedure's arity, name, and documentation. You can pick apart these pieces with the accessors in `(system vm program)`. See Section 6.9.3 [Compiled Procedures], page 238, for a full API reference.

The object array of a compiled procedure, also known as the *object table*, holds all Scheme objects whose values are known not to change across invocations of the procedure: constant strings, symbols, etc. The object table of a program is initialized right before a program is loaded with `load-program`. See Section 10.3.6.8 [Loading Instructions], page 752, for more information.

Variable objects are one such type of constant object: when a global binding is defined, a variable object is associated to it and that object will remain constant over time, even if the value bound to it changes. Therefore, toplevel bindings only need to be looked up once. Thereafter, references to the corresponding toplevel variables from within the program are then performed via the `toplevel-ref` instruction, which uses the object vector, and are almost as fast as local variable references.

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
   0    (assert-nargs-ee/locals 1)
   2    (object-ref 1)                  ;; #<procedure 8ebec20 at <current input>:0:17 (b)>
   4    (local-ref 0)                   ;; 'a'
   6    (make-closure 0 1)
   9    (return)

----------------------------------------
Disassembly of #<procedure 8ebec20 at <current input>:0:17 (b)>:

   0    (assert-nargs-ee/locals 1)
   2    (toplevel-ref 1)                ;; 'foo'
   4    (free-ref 0)                    ;; (closure variable)
   6    (local-ref 0)                   ;; 'b'
   8    (list 0 3)                      ;; 3 elements         at (unknown file):0:29
  11    (return)
```

First there's some prelude, where `foo` checks that it was called with only 1 argument. Then at `ip` 2, we load up the compiled lambda. `Ip` 4 loads up 'a', so that it can be captured into a closure by at `ip` 6—binding code (from the compiled lambda) with data (the free-variable vector). Finally we return the closure.

The second stanza disassembles the compiled lambda. After the prelude, we note that toplevel variables are resolved relative to the module that was current when the procedure was created. This lookup occurs lazily, at 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 10.3.6.2 [Top-Level Environment Instructions], page 745, for more details.

Then we see a reference to a free variable, corresponding to `a`. The disassembler doesn't have enough information to give a name to that variable, so it just marks it as being a "closure variable". Finally we see the reference to `b`, then the `list` opcode, an inline implementation of the `list` scheme routine.

10.3.6 Instruction Set

There are about 180 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 one byte long. Some instructions take parameters, which follow the instruction byte in the instruction stream.

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:

```
(if pred then else)
(if (not pred) then else)
(if (null? l) then else)
(if (not (null? l)) then else)
```

In addition, some Scheme primitives have their own inline implementations, e.g. `cons`, and `list`, as we saw in the previous section.

So Guile's instruction set is a *complete* instruction set, in that it provides the instructions that are suited to the problem, and is not concerned with making a minimal, orthogonal set of instructions. More instructions may be added over time.

10.3.6.1 Lexical Environment Instructions

These instructions access and mutate the lexical environment of a compiled procedure—its free and bound variables.

Some of these instructions have `long-` variants, the difference being that they take 16-bit arguments, encoded in big-endianness, instead of the normal 8-bit range.

See Section 10.3.3 [Stack Layout], page 740, for more information on the format of stack frames.

`local-ref` *index* [Instruction]
`long-local-ref` *index* [Instruction]

Push onto the stack the value of the local variable located at *index* within the current stack frame.

Note that arguments and local variables are all in one block. Thus the first argument, if any, is at index 0, and local bindings follow the arguments.

`local-set` *index* [Instruction]
`long-local-set` *index* [Instruction]

Pop the Scheme object located on top of the stack and make it the new value of the local variable located at *index* within the current stack frame.

`box` *index* [Instruction]

Pop a value off the stack, and set the *index*nth local variable to a box containing that value. A shortcut for `make-variable` then `local-set`, used when binding boxed variables.

`empty-box` *index* [Instruction]

Set the *index*th local variable to a box containing a variable whose value is unbound. Used when compiling some `letrec` expressions.

`local-boxed-ref` *index* [Instruction]
`local-boxed-set` *index* [Instruction]

Get or set the value of the variable located at *index* within the current stack frame. A shortcut for `local-ref` then `variable-ref` or `variable-set`, respectively.

`free-ref` *index* [Instruction]

Push the value of the captured variable located at position *index* within the program's vector of captured variables.

`free-boxed-ref` *index* [Instruction]
`free-boxed-set` *index* [Instruction]

Get or set a boxed free variable. A shortcut for `free-ref` then `variable-ref` or `variable-set`, respectively.

Note that there is no `free-set` instruction, as variables that are `set!` must be boxed.

`make-closure` *num-free-vars* [Instruction]

Pop *num-free-vars* values and a program object off the stack in that order, and push a new program object closing over the given free variables. *num-free-vars* is encoded as a two-byte big-endian value.

The free variables are stored in an array, inline to the new program object, in the order that they were on the stack (not the order they are popped off). The new closure shares state with the original program. At the time of this writing, the space overhead of closures is 3 words, plus one word for each free variable.

`fix-closure` *index* [Instruction]

Fix up the free variables array of the closure stored in the *index*th local variable. *index* is a two-byte big-endian integer.

This instruction will pop as many values from the stack as are in the corresponding closure's free variables array. The topmost value on the stack will be stored as the closure's last free variable, with other values filling in free variable slots in order.

`fix-closure` is part of a hack for allocating mutually recursive procedures. The hack is to store the procedures in their corresponding local variable slots, with space already allocated for free variables. Then once they are all in place, this instruction fixes up their procedures' free variable bindings in place. This allows most `letrec`-bound procedures to be allocated unboxed on the stack.

`local-bound?` *index* [Instruction]
`long-local-bound?` *index* [Instruction]

Push `#t` on the stack if the `index`th local variable has been assigned, or `#f` otherwise. Mostly useful for handling optional arguments in procedure prologues.

10.3.6.2 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.

Currently only toplevel references within procedures are cached, as only procedures have a place to cache them, in their object tables.

`toplevel-ref` *index* [Instruction]
`long-toplevel-ref` *index* [Instruction]

Push the value of the toplevel binding whose location is stored in at position *index* in the current procedure's object table. The `long-` variant encodes the index over two bytes.

Initially, a cell in a procedure's object table that is used by `toplevel-ref` is initialized to one of two forms. The normal case is that the cell holds a symbol, whose binding will be looked up relative to the module that was current when the current program was created.

Alternately, the lookup may be performed relative to a particular module, determined at compile-time (e.g. via `@` or `@@`). In that case, the cell in the object table holds a list: `(`*`modname sym public?`*`)`. The symbol *sym* will be looked up in the module named *modname* (a list of symbols). The lookup will be performed against the module's public interface, unless *public?* is `#f`, which it is for example when compiling `@@`.

In any case, if the symbol is unbound, an error is signalled. Otherwise the initial form is replaced with the looked-up variable, an in-place mutation of the object table. This mechanism provides for lazy variable resolution, and an important cached fast-path once the variable has been successfully resolved.

This instruction pushes the value of the variable onto the stack.

`toplevel-set` *index* [Instruction]
`long-toplevel-set` *index* [Instruction]

Pop a value off the stack, and set it as the value of the toplevel variable stored at *index* in the object table. If the variable has not yet been looked up, we do the lookup as in `toplevel-ref`.

`define` [Instruction]

Pop a symbol and a value from the stack, in that order. Look up its binding in the current toplevel environment, creating the binding if necessary. Set the variable to the value.

`link-now` [Instruction]

Pop a value, *x*, from the stack. Look up the binding for *x*, according to the rules for `toplevel-ref`, and push that variable on the stack. If the lookup fails, an error will be signalled.

This instruction is mostly used when loading programs, because it can do toplevel variable lookups without an object table.

`variable-ref` [Instruction]
Dereference the variable object which is on top of the stack and replace it by the value of the variable it represents.

`variable-set` [Instruction]
Pop off two objects from the stack, a variable and a value, and set the variable to the value.

`variable-bound?` [Instruction]
Pop off the variable object from top of the stack and push `#t` if it is bound, or `#f` otherwise. Mostly useful in procedure prologues for defining default values for boxed optional variables.

`make-variable` [Instruction]
Replace the top object on the stack with a variable containing it. Used in some circumstances when compiling `letrec` expressions.

10.3.6.3 Procedure Call and Return Instructions

`new-frame` [Instruction]
Push a new frame on the stack, reserving space for the dynamic link, return address, and the multiple-values return address. The frame pointer is not yet updated, because the frame is not yet active – it has to be patched by a `call` instruction to get the return address.

`call` *nargs* [Instruction]
Call the procedure located at `sp[-nargs]` with the *nargs* arguments located from `sp[-nargs + 1]` to `sp[0]`.

This instruction requires that a new frame be pushed on the stack before the procedure, via `new-frame`. See Section 10.3.3 [Stack Layout], page 740, for more information. It patches up that frame with the current `ip` as the return address, then dispatches to the first instruction in the called procedure, relying on the called procedure to return one value to the newly-created continuation. Because the new frame pointer will point to `sp[-nargs + 1]`, the arguments don't have to be shuffled around – they are already in place.

`tail-call` *nargs* [Instruction]
Transfer control to the procedure located at `sp[-nargs]` with the *nargs* arguments located from `sp[-nargs + 1]` to `sp[0]`.

Unlike `call`, which requires a new frame to be pushed onto the stack, `tail-call` simply shuffles down the procedure and arguments to the current stack frame. This instruction implements tail calls as required by RnRS.

`apply` *nargs* [Instruction]
`tail-apply` *nargs* [Instruction]
Like `call` and `tail-call`, except that the top item on the stack must be a list. The elements of that list are then pushed on the stack and treated as additional arguments, replacing the list itself, then the procedure is invoked as usual.

`call/nargs` [Instruction]
`tail-call/nargs` [Instruction]

These are like `call` and `tail-call`, except they take the number of arguments from the stack instead of the instruction stream. These instructions are used in the implementation of multiple value returns, where the actual number of values is pushed on the stack.

`mv-call` *nargs offset* [Instruction]

Like `call`, except that a multiple-value continuation is created in addition to a single-value continuation.

The offset (a three-byte value) is an offset within the instruction stream; the multiple-value return address in the new frame (see Section 10.3.3 [Stack Layout], page 740) will be set to the normal return address plus this offset. Instructions at that offset will expect the top value of the stack to be the number of values, and below that values themselves, pushed separately.

`return` [Instruction]

Free the program's frame, returning the top value from the stack to the current continuation. (The stack should have exactly one value on it.)

Specifically, the `sp` is decremented to one below the current `fp`, the `ip` is reset to the current return address, the `fp` is reset to the value of the current dynamic link, and then the returned value is pushed on the stack.

`return/values` *nvalues* [Instruction]
`return/nvalues` [Instruction]

Return the top *nvalues* to the current continuation. In the case of `return/nvalues`, *nvalues* itself is first popped from the top of the stack.

If the current continuation is a multiple-value continuation, `return/values` pushes the number of values on the stack, then returns as in `return`, but to the multiple-value return address.

Otherwise if the current continuation accepts only one value, i.e. the multiple-value return address is `NULL`, then we assume the user only wants one value, and we give them the first one. If there are no values, an error is signaled.

`return/values*` *nvalues* [Instruction]

Like a combination of `apply` and `return/values`, in which the top value on the stack is interpreted as a list of additional values. This is an optimization for the common `(apply values ...)` case.

`truncate-values` *nbinds nrest* [Instruction]

Used in multiple-value continuations, this instruction takes the values that are on the stack (including the number-of-values marker) and truncates them for a binding construct.

For example, a call to `(receive (x y . z) (foo) ...)` would, logically speaking, pop off the values returned from `(foo)` and push them as three values, corresponding to `x`, `y`, and `z`. In that case, *nbinds* would be 3, and *nrest* would be 1 (to indicate that one of the bindings was a rest argument).

Signals an error if there is an insufficient number of values.

`call/cc` [Instruction]
`tail-call/cc` [Instruction]

Capture the current continuation, and then call (or tail-call) the procedure on the top of the stack, with the continuation as the argument.

`call/cc` does not require a `new-frame` to be pushed on the stack, as `call` does, because it needs to capture the stack before the frame is pushed.

Both the VM continuation and the C continuation are captured.

10.3.6.4 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` *n* [Instruction]
`assert-nargs-ge` *n* [Instruction]

Assert that the current procedure has been passed exactly *n* arguments, for the `-ee` case, or *n* or more arguments, for the `-ge` case. *n* is encoded over two bytes.

The number of arguments is determined by subtracting the frame pointer from the stack pointer (`sp - (fp -1)`). See Section 10.3.3 [Stack Layout], page 740, for more details on stack frames.

`br-if-nargs-ne` *n offset* [Instruction]
`br-if-nargs-gt` *n offset* [Instruction]
`br-if-nargs-lt` *n offset* [Instruction]

Jump to *offset* if the number of arguments is not equal to, greater than, or less than *n*. *n* is encoded over two bytes, and *offset* has the normal three-byte encoding.

These instructions are used to implement multiple arities, as in `case-lambda`. See Section 6.9.5 [Case-lambda], page 243, for more information.

`bind-optionals` *n* [Instruction]

If the procedure has been called with fewer than *n* arguments, fill in the remaining arguments with an unbound value (`SCM_UNDEFINED`). *n* is encoded over two bytes.

The optionals can be later initialized conditionally via the `local-bound?` instruction.

`push-rest` *n* [Instruction]

Pop off excess arguments (more than *n*), collecting them into a list, and push that list. Used to bind a rest argument, if the procedure has no keyword arguments. Procedures with keyword arguments use `bind-rest` instead.

`bind-rest` *n idx* [Instruction]

Pop off excess arguments (more than *n*), collecting them into a list. The list is then assigned to the *idx*th local variable.

`bind-optionals/shuffle` *nreq nreq-and-opt ntotal* [Instruction]

Shuffle keyword arguments to the top of the stack, filling in the holes with `SCM_UNDEFINED`. Each argument is encoded over two bytes.

This instruction is used by procedures with keyword arguments. *nreq* is the number of required arguments to the procedure, and *nreq-and-opt* is the total number of positional arguments (required plus optional). `bind-optionals/shuffle` will scan the stack from the *nreq*th argument up to the *nreq-and-opt*th, and start shuffling when it sees the first keyword argument or runs out of positional arguments.

Shuffling simply moves the keyword arguments past the total number of arguments, *ntotal*, which includes keyword and rest arguments. The free slots created by the shuffle are filled in with `SCM_UNDEFINED`, so they may be conditionally initialized later in the function's prologue.

`bind-kwargs` *idx ntotal flags* [Instruction]

Parse keyword arguments, assigning their values to the corresponding local variables. The keyword arguments should already have been shuffled above the *ntotal*th stack slot by `bind-optionals/shuffle`.

The parsing is driven by a keyword arguments association list, looked up from the *idx*th element of the procedures object array. The alist is a list of pairs of the form `(kw . index)`, mapping keyword arguments to their local variable indices.

There are two bitflags that affect the parser, `allow-other-keys?` (`0x1`) and `rest?` (`0x2`). Unless `allow-other-keys?` is set, the parser will signal an error if an unknown key is found. If `rest?` is set, errors parsing the keyword arguments will be ignored, as a later `bind-rest` instruction will collect all of the tail arguments, including the keywords, into a list. Otherwise if the keyword arguments are invalid, an error is signalled.

idx and *ntotal* are encoded over two bytes each, and *flags* is encoded over one byte.

`reserve-locals` *n* [Instruction]

Resets the stack pointer to have space for *n* local variables, including the arguments. If this operation increments the stack pointer, as in a push, the new slots are filled with `SCM_UNBOUND`. If this operation decrements the stack pointer, any excess values are dropped.

`reserve-locals` is typically used after argument parsing to reserve space for local variables.

`assert-nargs-ee/locals` *n* [Instruction]
`assert-nargs-ge/locals` *n* [Instruction]

A combination of `assert-nargs-ee` and `reserve-locals`. The number of arguments is encoded in the lower three bits of *n*, a one-byte value. The number of additional local variables is take from the upper 5 bits of *n*.

10.3.6.5 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` *nargs* [Instruction]

Pop off a foreign pointer (which should have been pushed on by the trampoline), and call it directly, with the *nargs* arguments from the stack. Return the resulting value or values to the calling procedure.

`foreign-call` *nargs* [Instruction]

Pop off an internal foreign object (which should have been pushed on by the trampoline), and call that foreign function with the *nargs* arguments from the stack. Return the resulting value to the calling procedure.

`continuation-call` [Instruction]

Pop off an internal continuation object (which should have been pushed on by the trampoline), and reinstate that continuation. All of the procedure's arguments are passed to the continuation. Does not return.

`partial-cont-call` [Instruction]

Pop off two objects from the stack: the dynamic winds associated with the partial continuation, and the VM continuation object. Unroll the continuation onto the stack, rewinding the dynamic environment and overwriting the current frame, and pass all arguments to the continuation. Control flow proceeds where the continuation was captured.

10.3.6.6 Branch Instructions

All the conditional branch instructions described below work in the same way:

- They pop off Scheme object(s) located on the stack for use in the branch condition
- If the condition is true, then the instruction pointer is increased by the offset passed as an argument to the branch instruction;
- Program execution proceeds with the next instruction (that is, the one to which the instruction pointer points).

Note that the offset passed to the instruction is encoded as three 8-bit integers, in big-endian order, effectively giving Guile a 24-bit relative address space.

`br` *offset* [Instruction]

Jump to *offset*. No values are popped.

`br-if` *offset* [Instruction]

Jump to *offset* if the object on the stack is not false.

`br-if-not` *offset* [Instruction]

Jump to *offset* if the object on the stack is false.

`br-if-eq` *offset* [Instruction]

Jump to *offset* if the two objects located on the stack are equal in the sense of `eq?`. Note that, for this instruction, the stack pointer is decremented by two Scheme objects instead of only one.

`br-if-not-eq` *offset* [Instruction]

Same as `br-if-eq` for non-`eq?` objects.

`br-if-null` *offset* [Instruction]

Jump to *offset* if the object on the stack is `'()`.

`br-if-not-null` *offset* [Instruction]

Jump to *offset* if the object on the stack is not `'()`.

10.3.6.7 Data Constructor Instructions

These instructions push simple immediate values onto the stack, or construct compound data structures from values on the stack.

`make-int8` *value* [Instruction]

Push *value*, an 8-bit integer, onto the stack.

`make-int8:0` [Instruction]

Push the immediate value `0` onto the stack.

`make-int8:1` [Instruction]

Push the immediate value `1` onto the stack.

`make-int16` *value* [Instruction]

Push *value*, a 16-bit integer, onto the stack.

`make-uint64` *value* [Instruction]

Push *value*, an unsigned 64-bit integer, onto the stack. The value is encoded in 8 bytes, most significant byte first (big-endian).

`make-int64` *value* [Instruction]

Push *value*, a signed 64-bit integer, onto the stack. The value is encoded in 8 bytes, most significant byte first (big-endian), in twos-complement arithmetic.

`make-false` [Instruction]

Push `#f` onto the stack.

`make-true` [Instruction]

Push `#t` onto the stack.

`make-nil` [Instruction]

Push `#nil` onto the stack.

`make-eol` [Instruction]

Push `'()` onto the stack.

`make-char8` *value* [Instruction]

Push *value*, an 8-bit character, onto the stack.

`make-char32` *value* [Instruction]

Push *value*, an 32-bit character, onto the stack. The value is encoded in big-endian order.

`make-symbol` [Instruction]

Pops a string off the stack, and pushes a symbol.

`make-keyword` *value* [Instruction]

Pops a symbol off the stack, and pushes a keyword.

`list` *n* [Instruction]

Pops off the top *n* values off of the stack, consing them up into a list, then pushes that list on the stack. What was the topmost value will be the last element in the list. *n* is a two-byte value, most significant byte first.

`vector` *n* [Instruction]

Create and fill a vector with the top *n* values from the stack, popping off those values and pushing on the resulting vector. *n* is a two-byte value, like in `vector`.

`make-struct` *n* [Instruction]

Make a new struct from the top *n* values on the stack. The values are popped, and the new struct is pushed.

The deepest value is used as the vtable for the struct, and the rest are used in order as the field initializers. Tail arrays are not supported by this instruction.

`make-array` *n* [Instruction]

Pop an array shape from the stack, then pop the remaining *n* values, pushing a new array. *n* is encoded over three bytes.

The array shape should be appropriate to store *n* values. See Section 6.7.6.2 [Array Procedures], page 199, for more information on array shapes.

Many of these data structures are constant, never changing over the course of the different invocations of the procedure. In that case it is often advantageous to make them once when the procedure is created, and just reference them from the object table thereafter. See Section 10.3.4 [Variables and the VM], page 741, for more information on the object table.

`object-ref` *n* [Instruction]
`long-object-ref` *n* [Instruction]

Push *n*th value from the current program's object vector. The "long" variant has a 16-bit index instead of an 8-bit index.

10.3.6.8 Loading Instructions

In addition to VM instructions, an instruction stream may contain variable-length data embedded within it. This data is always preceded by special loading instructions, which interpret the data and advance the instruction pointer to the next VM instruction.

All of these loading instructions have a `length` parameter, indicating the size of the embedded data, in bytes. The length itself is encoded in 3 bytes.

`load-number` *length* [Instruction]

Load an arbitrary number from the instruction stream. The number is embedded in the stream as a string.

`load-string` *length* [Instruction]

Load a string from the instruction stream. The string is assumed to be encoded in the "latin1" locale.

`load-wide-string` *length* [Instruction]

Load a UTF-32 string from the instruction stream. *length* is the length in bytes, not in codepoints.

`load-symbol` *length* [Instruction]

Load a symbol from the instruction stream. The symbol is assumed to be encoded in the "latin1" locale. Symbols backed by wide strings may be loaded via `load-wide-string` then `make-symbol`.

`load-array` *length* [Instruction]

Load a uniform array from the instruction stream. The shape and type of the array are popped off the stack, in that order.

`load-program` [Instruction]

Load bytecode from the instruction stream, and push a compiled procedure.

This instruction pops one value from the stack: the program's object table, as a vector, or `#f` in the case that the program has no object table. A program that does not reference toplevel bindings and does not use `object-ref` does not need an object table.

This instruction is unlike the rest of the loading instructions, because instead of parsing its data, it directly maps the instruction stream onto a C structure, `struct scm_objcode`. See Section 10.4.6 [Bytecode and Objcode], page 769, for more information.

The resulting compiled procedure will not have any free variables captured, so it may be loaded only once but used many times to create closures.

10.3.6.9 Dynamic Environment Instructions

Guile's virtual machine has low-level support for `dynamic-wind`, dynamic binding, and composable prompts and aborts.

`wind` [Instruction]

Pop an unwind thunk and a wind thunk from the stack, in that order, and push them onto the "dynamic stack". The unwind thunk will be called on nonlocal exits, and the wind thunk on reentries. Used to implement `dynamic-wind`.

Note that neither thunk is actually called; the compiler should emit calls to wind and unwind for the normal dynamic-wind control flow. See Section 6.13.10 [Dynamic Wind], page 298.

`unwind` [Instruction]

Pop off the top entry from the "dynamic stack", for example, a wind/unwind thunk pair. `unwind` instructions should be properly paired with their winding instructions, like `wind`.

`wind-fluids` *n* [Instruction]

Pop off *n* values and *n* fluids from the stack, in that order. Set the fluids to the values by creating a with-fluids object and pushing that object on the dynamic stack. See Section 6.21.7 [Fluids and Dynamic States], page 405.

`unwind-fluids` [Instruction]

Pop a with-fluids object from the dynamic stack, and swap the current values of its fluids with the saved values of its fluids. In this way, the dynamic environment is left as it was before the corresponding `wind-fluids` instruction was processed.

`fluid-ref` [Instruction]

Pop a fluid from the stack, and push its current value.

`fluid-set` [Instruction]

Pop a value and a fluid from the stack, in that order, and set the fluid to the value.

`prompt` *escape-only? offset* [Instruction]

Establish a dynamic prompt. See Section 6.13.5 [Prompts], page 285, for more information on prompts.

The prompt will be pushed on the dynamic stack. The normal control flow should ensure that the prompt is popped off at the end, via `unwind`.

If an abort is made to this prompt, control will jump to *offset*, a three-byte relative address. The continuation and all arguments to the abort will be pushed on the stack, along with the total number of arguments (including the continuation. 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.

`abort` *n* [Instruction]

Abort to a dynamic prompt.

This instruction pops one tail argument list, *n* arguments, and a prompt tag from the stack. The dynamic environment is then searched for a prompt having the given tag. If none is found, 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, along with the number of arguments, as in the multiple-value return convention. Therefore an `abort` instruction should be followed by code ready to handle the equivalent of a multiply-valued return.

10.3.6.10 Miscellaneous Instructions

`nop` [Instruction]

Does nothing! Used for padding other instructions to certain alignments.

`halt` [Instruction]

Exits the VM, returning a SCM value. Normally, this instruction is only part of the "bootstrap program", a program run when a virtual machine is first entered; compiled Scheme procedures will not contain this instruction.

If multiple values have been returned, the SCM value will be a multiple-values object (see Section 6.13.7 [Multiple Values], page 290).

`break` [Instruction]

Does nothing, but invokes the break hook.

`drop` [Instruction]

Pops off the top value from the stack, throwing it away.

`dup` [Instruction]

Re-pushes the top value onto the stack.

`void` [Instruction]

Pushes "the unspecified value" onto the stack.

10.3.6.11 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.

Since most of these operations are historically implemented as C primitives, not inlining them would entail constantly calling out from the VM to the interpreter, which has some costs—registers must be saved, the interpreter has to dispatch, called procedures have to do much type checking, etc. It's much more efficient to inline these operations in the virtual machine itself.

All of these instructions pop their arguments from the stack and push their results, and take no parameters from the instruction stream. Thus, unlike in the previous sections, these instruction definitions show stack parameters instead of parameters from the instruction stream.

`not` *x* [Instruction]
`not-not` *x* [Instruction]
`eq?` *x y* [Instruction]
`not-eq?` *x y* [Instruction]
`null?` [Instruction]
`not-null?` [Instruction]
`eqv?` *x y* [Instruction]
`equal?` *x y* [Instruction]
`pair?` *x y* [Instruction]
`list?` *x* [Instruction]
`set-car!` *pair x* [Instruction]
`set-cdr!` *pair x* [Instruction]
`cons` *x y* [Instruction]
`car` *x* [Instruction]
`cdr` *x* [Instruction]
`vector-ref` *x y* [Instruction]

`vector-set` *x n y* [Instruction]
`struct?` *x* [Instruction]
`struct-ref` *x n* [Instruction]
`struct-set` *x n v* [Instruction]
`struct-vtable` *x* [Instruction]
`class-of` *x* [Instruction]
`slot-ref` *struct n* [Instruction]
`slot-set` *struct n x* [Instruction]

Inlined implementations of their Scheme equivalents.

Note that `caddr` and friends compile to a series of `car` and `cdr` instructions.

10.3.6.12 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.

As in the previous section, the definitions below show stack parameters instead of instruction stream parameters.

`add` *x y* [Instruction]
`add1` *x* [Instruction]
`sub` *x y* [Instruction]
`sub1` *x* [Instruction]
`mul` *x y* [Instruction]
`div` *x y* [Instruction]
`quo` *x y* [Instruction]
`rem` *x y* [Instruction]
`mod` *x y* [Instruction]
`ee?` *x y* [Instruction]
`lt?` *x y* [Instruction]
`gt?` *x y* [Instruction]
`le?` *x y* [Instruction]
`ge?` *x y* [Instruction]
`ash` *x n* [Instruction]
`logand` *x y* [Instruction]
`logior` *x y* [Instruction]
`logxor` *x y* [Instruction]

Inlined implementations of the corresponding mathematical operations.

10.3.6.13 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.

As in the previous section, the definitions below show stack parameters instead of instruction stream parameters.

The multibyte formats (`u16`, `f64`, etc) take an extra endianness argument. Only aligned native accesses are currently fast-pathed in Guile's VM.

`bv-u8-ref` *bv n* [Instruction]
`bv-s8-ref` *bv n* [Instruction]
`bv-u16-native-ref` *bv n* [Instruction]
`bv-s16-native-ref` *bv n* [Instruction]
`bv-u32-native-ref` *bv n* [Instruction]
`bv-s32-native-ref` *bv n* [Instruction]
`bv-u64-native-ref` *bv n* [Instruction]
`bv-s64-native-ref` *bv n* [Instruction]
`bv-f32-native-ref` *bv n* [Instruction]
`bv-f64-native-ref` *bv n* [Instruction]
`bv-u16-ref` *bv n endianness* [Instruction]
`bv-s16-ref` *bv n endianness* [Instruction]
`bv-u32-ref` *bv n endianness* [Instruction]
`bv-s32-ref` *bv n endianness* [Instruction]
`bv-u64-ref` *bv n endianness* [Instruction]
`bv-s64-ref` *bv n endianness* [Instruction]
`bv-f32-ref` *bv n endianness* [Instruction]
`bv-f64-ref` *bv n endianness* [Instruction]
`bv-u8-set` *bv n val* [Instruction]
`bv-s8-set` *bv n val* [Instruction]
`bv-u16-native-set` *bv n val* [Instruction]
`bv-s16-native-set` *bv n val* [Instruction]
`bv-u32-native-set` *bv n val* [Instruction]
`bv-s32-native-set` *bv n val* [Instruction]
`bv-u64-native-set` *bv n val* [Instruction]
`bv-s64-native-set` *bv n val* [Instruction]
`bv-f32-native-set` *bv n val* [Instruction]
`bv-f64-native-set` *bv n val* [Instruction]
`bv-u16-set` *bv n val endianness* [Instruction]
`bv-s16-set` *bv n val endianness* [Instruction]
`bv-u32-set` *bv n val endianness* [Instruction]
`bv-s32-set` *bv n val endianness* [Instruction]
`bv-u64-set` *bv n val endianness* [Instruction]
`bv-s64-set` *bv n val endianness* [Instruction]
`bv-f32-set` *bv n val endianness* [Instruction]
`bv-f64-set` *bv n val endianness* [Instruction]

Inlined implementations of the corresponding bytevector operations.

10.4 Compiling to the Virtual Machine

Compilers have a mystique about them that is attractive and off-putting at the same time. They are attractive because they are magical – they transform inert text into live results, like throwing the switch on Frankenstein's monster. However, this magic is perceived by many to be impenetrable.

This section aims to pay attention to the small man behind the curtain.

See Section 6.17 [Read/Load/Eval/Compile], page 343, if you're lost and you just wanted to know how to compile your `.scm` file.

10.4.1 Compiler Tower

Guile's compiler is quite simple, actually – its *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 10.3.6 [Instruction Set], page 743).

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*] [*make-default-environment=make-fresh-user-module*] [Scheme Syntax]

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* [Scheme Procedure]

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* [Scheme Procedure]

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*` fluid. This language is normally Scheme, and may be rebound by the user. The run-time compilation interfaces (see Section 6.17 [Read/Load/Eval/Compile], page 343) 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)
- Guile Lowlevel Intermediate Language (GLIL)
- Assembly
- Bytecode
- Objcode

Object code may be serialized to disk directly, though it has a cookie and version prepended to the front. 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 object code into a procedure, and wakes the sleeping giant.

Perhaps this strangeness can be explained by example: `compile-file` defaults to compiling to object code, 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)))
```

10.4.2 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" is given by the inventory of constructs provided by Tree-IL, the target language of the Scheme compiler: procedure applications, conditionals, lexical references, etc. 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)
⇒
 #<<application> src: #f
                 proc: #<<toplevel-ref> src: #f name: +>
                 args: (#<<const> src: #f exp: 1>
                        #<<const> src: #f exp: 2>)>
```

Or, since Tree-IL is so close to Scheme, it is often useful to expand Scheme to Tree-IL, then translate back to Scheme. For that reason the expander provides two interfaces. The former is equivalent to calling `(macroexpand '(+ 1 2) 'c)`, where the `'c` is for "compile". With `'e` (the default), the result is translated back to Scheme:

```
(macroexpand '(+ 1 2))
⇒ (+ 1 2)
(macroexpand '(let ((x 10)) (* x x)))
⇒ (let ((x84 10)) (* x84 x84))
```

The second example shows that as part of its job, the macro expander renames lexically-bound variables. The original names are preserved when compiling to Tree-IL, but can't be represented in Scheme: a lexical binding only has one name. It is for this reason that the *native* output of the expander is *not* Scheme. There's too much information we would lose if we translated to Scheme directly: lexical variable names, source locations, and module hygiene.

Note however that `macroexpand` does not have the same signature as `compile-tree-il`. `compile-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 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 6.17.6 [Loading], page 352) 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
```

10.4.3 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 6.7.8 [Records], page 212, 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 6.25.2 [Source Properties], page 432, 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)> (apply (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>` *src* [Scheme Variable]
`(void)` [External Representation]

An empty expression. In practice, equivalent to Scheme's `(if #f #f)`.

`<const>` *src exp* [Scheme Variable]
`(const` *exp*`)` [External Representation]

A constant.

`<primitive-ref>` *src name* [Scheme Variable]
`(primitive` *name*`)` [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 applications of certain primitives, like `apply` or `cons`.

`<lexical-ref>` *src name gensym* [Scheme Variable]
`(lexical` *name gensym*`)` [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* [Scheme Variable]
`(set! (`*lexical* *name gensym*`) `*exp*`)` [External Representation]

Sets a lexically-bound variable.

`<module-ref>` *src mod name public?* [Scheme Variable]
`(@` *mod name*`)` [External Representation]
`(@@` *mod name*`)` [External Representation]

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* [Scheme Variable]
`(set! (@` *mod name*`) `*exp*`)` [External Representation]
`(set! (@@` *mod name*`) `*exp*`)` [External Representation]

Sets a variable in a specific module.

`<toplevel-ref>` *src name* [Scheme Variable]
`(toplevel name)` [External Representation]

References a variable from the current procedure's module.

`<toplevel-set>` *src name exp* [Scheme Variable]
`(set! (toplevel name) exp)` [External Representation]

Sets a variable in the current procedure's module.

`<toplevel-define>` *src name exp* [Scheme Variable]
`(define (toplevel name) exp)` [External Representation]

Defines a new top-level variable in the current procedure's module.

`<conditional>` *src test then else* [Scheme Variable]
`(if test then else)` [External Representation]

A conditional. Note that *else* is not optional.

`<application>` *src proc args* [Scheme Variable]
`(apply proc . args)` [External Representation]

A procedure call.

`<sequence>` *src exps* [Scheme Variable]
`(begin . exps)` [External Representation]

Like Scheme's `begin`.

`<lambda>` *src meta body* [Scheme Variable]
`(lambda meta body)` [External Representation]

A closure. *meta* is an association list of properties for the procedure. *body* is a single Tree-IL expression of type `<lambda-case>`. As the `<lambda-case>` clause can chain to an alternate clause, this makes Tree-IL's `<lambda>` have the expressiveness of Scheme's `case-lambda`.

`<lambda-case>` *req opt rest kw inits gensyms body alternate* [Scheme Variable]
`(lambda-case ((req opt rest kw inits gensyms) body) [alternate])` [External Representation]

One clause of a `case-lambda`. A `lambda` expression in Scheme is treated as a `case-lambda` with one clause.

req is a list of the procedure's required arguments, as symbols. *opt* is a list of the optional arguments, or `#f` if there are no optional arguments. *rest* is the name of the rest argument, or `#f`.

kw is a list of the form, `(allow-other-keys? (keyword name var) ...)`, where *keyword* is the keyword corresponding to the argument named *name*, and whose corresponding gensym is *var*. *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 *init* expression is evaluated in the lexical context of previously bound variables, from left to right.

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.

body is the body of the clause. If the procedure is called with an appropriate number of arguments, *body* is evaluated in tail position. Otherwise, if there is an *alternate*, it should be a `<lambda-case>` expression, representing the next clause to try. If there is no *alternate*, a wrong-number-of-arguments error is signaled.

`<let>` *src names gensyms vals exp* [Scheme Variable]
`(let` ***names gensyms vals exp***`)` [External Representation]

Lexical binding, like Scheme's `let`. *names* are the original binding names, *gensyms* are gensyms corresponding to the *names*, and *vals* are Tree-IL expressions for the values. *exp* is a single Tree-IL expression.

`<letrec>` *in-order? src names gensyms vals exp* [Scheme Variable]
`(letrec` ***names gensyms vals exp***`)` [External Representation]
`(letrec*` ***names gensyms vals exp***`)` [External Representation]

A version of `<let>` that creates recursive bindings, like Scheme's `letrec`, or `letrec*` if *in-order?* is true.

`<dynlet>` *fluids vals body* [Scheme Variable]
`(dynlet` ***fluids vals body***`)` [External Representation]

Dynamic binding; the equivalent of Scheme's `with-fluids`. *fluids* should be a list of Tree-IL expressions that will evaluate to fluids, and *vals* a corresponding list of expressions to bind to the fluids during the dynamic extent of the evaluation of *body*.

`<dynref>` *fluid* [Scheme Variable]
`(dynref` ***fluid***`)` [External Representation]

A dynamic variable reference. *fluid* should be a Tree-IL expression evaluating to a fluid.

`<dynset>` *fluid exp* [Scheme Variable]
`(dynset` ***fluid exp***`)` [External Representation]

A dynamic variable set. *fluid*, a Tree-IL expression evaluating to a fluid, will be set to the result of evaluating *exp*.

`<dynwind>` *winder body unwinder* [Scheme Variable]
`(dynwind` ***winder body unwinder***`)` [External Representation]

A `dynamic-wind`. *winder* and *unwinder* should both evaluate to thunks. Ensure that the winder and the unwinder are called before entering and after leaving *body*. Note that *body* is an expression, without a thunk wrapper.

`<prompt>` *tag body handler* [Scheme Variable]
`(prompt` ***tag body handler***`)` [External Representation]

A dynamic prompt. Instates a prompt named *tag*, an expression, during the dynamic extent of the execution of *body*, also an expression. If an abort occurs to this prompt, control will be passed to *handler*, a `<lambda-case>` expression with no optional or keyword arguments, and no alternate. The first argument to the `<lambda-case>` will be the captured continuation, and then all of the values passed to the abort. See Section 6.13.5 [Prompts], page 285, for more information.

`<abort>` *tag args tail* [Scheme Variable]
`(abort` *tag args tail*`)` [External Representation]

An abort to the nearest prompt with the name *tag*, an expression. *args* should be a list of expressions to pass to the prompt's handler, and *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 `<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.

`<let-values>` *src names gensyms exp body* [Scheme Variable]
`(let-values` *names gensyms exp body*`)` [External Representation]

Like Scheme's `receive` – binds the values returned by evaluating `exp` to the `lambda`-like bindings described by *gensyms*. That is to say, *gensyms* may be an improper list.

`<let-values>` is an optimization of `<application>` of the primitive, `call-with-values`.

`<fix>` *src names gensyms vals body* [Scheme Variable]
`(fix` *names gensyms vals body*`)` [External Representation]

Like `<letrec>`, but only for *vals* that are unset `lambda` expressions.

`fix` is an optimization of `letrec` (and `let`).

Tree-IL implements a compiler to GLIL that recursively traverses Tree-IL expressions, writing out GLIL expressions into a linear list. The compiler also keeps some state as to whether the current expression is in tail context, and whether its value will be used in future computations. This state allows the compiler not to emit code for constant expressions that will not be used (e.g. docstrings), and to perform tail calls when in tail position.

Most optimization, such as it currently is, is performed on Tree-IL expressions as source-to-source transformations. There will be more optimizations added in the future.

Interested readers are encouraged to read the implementation in `(language tree-il compile-glil)` for more details.

10.4.4 GLIL

Guile Lowlevel Intermediate Language (GLIL) is a structured intermediate language whose expressions more closely approximate Guile's VM instruction set. Its expression types are defined in `(language glil)`.

`<glil-program>` *meta . body* [Scheme Variable]

A unit of code that at run-time will correspond to a compiled procedure. *meta* should be an alist of properties, as in Tree-IL's `<lambda>`. *body* is an ordered list of GLIL expressions.

`<glil-std-prelude>` *nreq nlocs else-label* [Scheme Variable]

A prologue for a function with no optional, keyword, or rest arguments. *nreq* is the number of required arguments. *nlocs* the total number of local variables, including

the arguments. If the procedure was not given exactly *nreq* arguments, control will jump to *else-label*, if given, or otherwise signal an error.

`<glil-opt-prelude>` *nreq nopt rest nlocs else-label* [Scheme Variable]

A prologue for a function with optional or rest arguments. Like `<glil-std-prelude>`, with the addition that *nopt* is the number of optional arguments (possibly zero) and *rest* is an index of a local variable at which to bind a rest argument, or `#f` if there is no rest argument.

`<glil-kw-prelude>` *nreq nopt rest kw allow-other-keys? nlocs else-label* [Scheme Variable]

A prologue for a function with keyword arguments. Like `<glil-opt-prelude>`, with the addition that *kw* is a list of keyword arguments, and *allow-other-keys?* is a flag indicating whether to allow unknown keys. See Section 10.3.6.4 [Function Prologue Instructions], page 748, for details on the format of *kw*.

`<glil-bind>` . *vars* [Scheme Variable]

An advisory expression that notes a liveness extent for a set of variables. *vars* is a list of `(name type index)`, where *type* should be either `argument`, `local`, or `external`.

`<glil-bind>` expressions end up being serialized as part of a program's metadata and do not form part of a program's code path.

`<glil-mv-bind>` *vars rest* [Scheme Variable]

A multiple-value binding of the values on the stack to *vars*. Iff *rest* is true, the last element of *vars* will be treated as a rest argument.

In addition to pushing a binding annotation on the stack, like `<glil-bind>`, an expression is emitted at compilation time to make sure that there are enough values available to bind. See the notes on `truncate-values` in Section 10.3.6.3 [Procedure Call and Return Instructions], page 746, for more information.

`<glil-unbind>` [Scheme Variable]

Closes the liveness extent of the most recently encountered `<glil-bind>` or `<glil-mv-bind>` expression. As GLIL expressions are compiled, a parallel stack of live bindings is maintained; this expression pops off the top element from that stack.

Bindings are written into the program's metadata so that debuggers and other tools can determine the set of live local variables at a given offset within a VM program.

`<glil-source>` *loc* [Scheme Variable]

Records source information for the preceding expression. *loc* should be an association list of containing `line column`, and `filename` keys, e.g. as returned by `source-properties`.

`<glil-void>` [Scheme Variable]

Pushes "the unspecified value" on the stack.

`<glil-const>` *obj* [Scheme Variable]

Pushes a constant value onto the stack. *obj* must be a number, string, symbol, keyword, boolean, character, uniform array, the empty list, or a pair or vector of constants.

`<glil-lexical>` *local? boxed? op index* [Scheme Variable]

Accesses a lexically bound variable. If the variable is not *local?* it is free. All variables may have `ref`, `set`, and `bound?` as their *op*. Boxed variables may also have the *ops* `box`, `empty-box`, and `fix`, which correspond in semantics to the VM instructions `box`, `empty-box`, and `fix-closure`. See Section 10.3.3 [Stack Layout], page 740, for more information.

`<glil-toplevel>` *op name* [Scheme Variable]

Accesses a toplevel variable. *op* may be `ref`, `set`, or `define`.

`<glil-module>` *op mod name public?* [Scheme Variable]

Accesses a variable within a specific module. See Tree-IL's `<module-ref>`, for more information.

`<glil-label>` *label* [Scheme Variable]

Creates a new label. *label* can be any Scheme value, and should be unique.

`<glil-branch>` *inst label* [Scheme Variable]

Branch to a label. *label* should be a `<ghil-label>`. `inst` is a branching instruction: `br-if`, `br`, etc.

`<glil-call>` *inst nargs* [Scheme Variable]

This expression is probably misnamed, as it does not correspond to function calls. `<glil-call>` invokes the VM instruction named *inst*, noting that it is called with *nargs* stack arguments. The arguments should be pushed on the stack already. What happens to the stack afterwards depends on the instruction.

`<glil-mv-call>` *nargs ra* [Scheme Variable]

Performs a multiple-value call. *ra* is a `<glil-label>` corresponding to the multiple-value return address for the call. See the notes on `mv-call` in Section 10.3.6.3 [Procedure Call and Return Instructions], page 746, for more information.

`<glil-prompt>` *label escape-only?* [Scheme Variable]

Push a dynamic prompt into the stack, with a handler at *label*. *escape-only?* is a flag that is propagated to the prompt, allowing an abort to avoid capturing a continuation in some cases. See Section 6.13.5 [Prompts], page 285, for more information.

Users may enter in GLIL at the REPL as well, though there is a bit more bookkeeping to do:

```
scheme@(guile-user)> ,language glil
Happy hacking with Guile Lowlevel Intermediate Language (GLIL)!
To switch back, type ',L scheme'.
glil@(guile-user)> (program () (std-prelude 0 0 #f)
                       (const 3) (call return 1))
⇒ 3
```

Just as in all of Guile's compilers, an environment is passed to the GLIL-to-object code compiler, and one is returned as well, along with the object code.

10.4.5 Assembly

Assembly is an S-expression-based, human-readable representation of the actual bytecodes that will be emitted for the VM. As such, it is a useful intermediate language both for compilation and for decompilation.

Besides the fact that it is not a record-based language, assembly differs from GLIL in four main ways:

- Labels have been resolved to byte offsets in the program.
- Constants inside procedures have either been expressed as inline instructions or cached in object arrays.
- Procedures with metadata (source location information, liveness extents, procedure names, generic properties, etc) have had their metadata serialized out to thunks.
- All expressions correspond directly to VM instructions – i.e., there is no `<glil-lexical>` which can be a ref or a set.

Assembly is isomorphic to the bytecode that it compiles to. You can compile to bytecode, then decompile back to assembly, and you have the same assembly code.

The general form of assembly instructions is the following:

```
(inst arg ...)
```

The *inst* names a VM instruction, and its *arg*s will be embedded in the instruction stream. The easiest way to see assembly is to play around with it at the REPL, as can be seen in this annotated example:

```
scheme@(guile-user)> ,pp (compile '(+ 32 10) #:to 'assembly)
(load-program
  ((:LCASE16 . 2))  ; Labels, unused in this case.
  8                 ; Length of the thunk that was compiled.
  (load-program     ; Metadata thunk.
    ()
    17
    #f              ; No metadata thunk for the metadata thunk.
    (make-eol)
    (make-eol)
    (make-int8 2)   ; Liveness extents, source info, and arities,
    (make-int8 8)   ; in a format that Guile knows how to parse.
    (make-int8:0)
    (list 0 3)
    (list 0 1)
    (list 0 3)
    (return))
  (assert-nargs-ee/locals 0)  ; Prologue.
  (make-int8 32)    ; Actual code starts here.
  (make-int8 10)
  (add)
  (return))
```

Of course you can switch the REPL to assembly and enter in assembly S-expressions directly, like with other languages, though it is more difficult, given that the length fields have to be correct.

10.4.6 Bytecode and Objcode

Finally, the raw bytes. There are actually two different "languages" here, corresponding to two different ways to represent the bytes.

"Bytecode" represents code as uniform byte vectors, useful for structuring and destructuring code on the Scheme level. Bytecode is the next step down from assembly:

```
scheme@(guile-user)> (compile '(+ 32 10) #:to 'bytecode)
⇒ #vu8(8 0 0 0 25 0 0 0            ; Header.
       95 0                           ; Prologue.
       10 32 10 10 148 66 17          ; Actual code.
       0 0 0 0 0 0 0 9                ; Metadata thunk.
       9 10 2 10 8 11 18 0 3 18 0 1 18 0 3 66)
```

"Objcode" is bytecode, but mapped directly to a C structure, `struct scm_objcode`:

```
struct scm_objcode {
  scm_t_uint32 len;
  scm_t_uint32 metalen;
  scm_t_uint8 base[0];
};
```

As one might imagine, objcode imposes a minimum length on the bytecode. Also, the `len` and `metalen` fields are in native endianness, which makes objcode (and bytecode) system-dependent.

Objcode also has a couple of important efficiency hacks. First, objcode may be mapped directly from disk, allowing compiled code to be loaded quickly, often from the system's disk cache, and shared among multiple processes. Secondly, objcode may be embedded in other objcode, allowing procedures to have the text of other procedures inlined into their bodies, without the need for separate allocation of the code. Of course, the objcode object itself does need to be allocated.

Procedures related to objcode are defined in the `(system vm objcode)` module.

`objcode?` *obj* [Scheme Procedure]
`scm_objcode_p` (*obj*) [C Function]

Returns `#f` iff *obj* is object code, `#f` otherwise.

`bytecode->objcode` *bytecode* [Scheme Procedure]
`scm_bytecode_to_objcode` (*bytecode*) [C Function]

Makes a bytecode object from *bytecode*, which should be a bytevector. See Section 6.6.6 [Bytevectors], page 162.

`load-objcode` *file* [Scheme Variable]
`scm_load_objcode` (*file*) [C Function]

Load object code from a file named *file*. The file will be mapped into memory via `mmap`, so this is a very fast operation.

On disk, object code has an sixteen-byte cookie prepended to it, to prevent accidental loading of arbitrary garbage.

`write-objcode` *objcode file* [Scheme Variable]
`scm_write_objcode` (*objcode*) [C Function]
Write object code out to a file, prepending the sixteen-byte cookie.

`objcode->bytecode` *objcode* [Scheme Variable]
`scm_objcode_to_bytecode` (*objcode*) [C Function]
Copy object code out to a bytevector for analysis by Scheme.

The following procedure is actually in `(system vm program)`, but we'll mention it here:

`make-program` *objcode objtable* [*free-vars=#f*] [Scheme Variable]
`scm_make_program` (*objcode, objtable, free_vars*) [C Function]
Load up object code into a Scheme program. The resulting program will have *objtable* as its object table, which should be a vector or `#f`, and will capture the free variables from *free-vars*.

Object code from a file may be disassembled at the REPL via the meta-command `,disassemble-file`, abbreviated as `,xx`. Programs may be disassembled via `,disassemble`, abbreviated as `,x`.

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.

10.4.7 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.

10.4.8 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

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
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-Schemey names for several features, to make it easier to locate the desired sections.

!

!# ... 345

#

#! ... 345
#,() ... 560

(

(ice-9 match) ... 626

.

.guile ... 45
'`.guile`' file, not loading ... 37
'`.guile_history`' ... 629
'`.inputrc`' ... 629

/

'`/etc/hosts`' ... 488
'`/etc/protocols`' ... 490
'`/etc/services`' ... 491

A

`addrinfo` object type ... 485
affinity, CPU ... 478
alist ... 220, 544
arbiters ... 396
argument specialize ... 575
arguments (command line) ... 35
arity, variable ... 243, 562
association list ... 544
association List ... 220
asyncs ... 397
atomic time ... 568
autoload ... 368
automatic compilation ... 350
automatically-managed memory ... 359

B

begin ... 280
binary input ... 326, 327
binary output ... 331
binary port ... 321
binding renamer ... 366
`bindir` ... 413
bitwise logical ... 586
block comments ... 345
Block read/write ... 312
Breakpoints ... 440, 447
Buffered input ... 652
`buildstamp` ... 413
byte order ... 162
bytevector ... 162

C

C port interface ... 332
callbacks ... 395
canonical host type ... 414
case ... 281
case folding ... 131
chaining environments ... 27
charset ... 138
child processes ... 473
class ... 685
Closing ports ... 309
closure ... 26
code coverage ... 448
Code coverage ... 440
code point ... 129
code point, designated ... 129
code point, reserved ... 129
codec ... 321
codeset ... 138
command line ... 471, 472
command line history ... 628
command-line arguments ... 35
Command-line Options ... 35
commands ... 47
composable continuations ... 285
cond ... 281
condition variable ... 400
conditional evaluation ... 281
conditions ... 578
conservative garbage collection ... 359
continuations ... 288
copying ... 1
coverage ... 448
cultural conventions ... 421
current directory ... 473
custom binary input ports ... 327
custom binary output ports ... 331

D

database ... 220
`datadir` ... 413
date ... 568, 570

date conversion ... 571
date to string ... 572
date, from string ... 574
debug options ... 438
Debugging ... 429
debugging virtual machine (command line) ... 37
Default ports ... 313
default slot value ... 687
definition splicing ... 280
delayed evaluation ... 356
delimited continuations ... 285
designated code point ... 129
device file ... 465
directory contents ... 464
directory traversal ... 644
domain-specific language ... 248
DSL ... 248
duplicate binding ... 368, 369
duplicate binding handlers ... 369

E

EDSL ... 248
effective version ... 57
Emacs ... 53
emacs regexp ... 335
embedded domain-specific language ... 248
encapsulation ... 365
encoding ... 356
encryption ... 501
End of file object ... 306
end-of-file ... 324
end-of-line style ... 321
endianness ... 162
environment ... 26, 381, 472
environment variables ... 38, 591
environment, local ... 27
environment, top level ... 26
EOF ... 324
equality ... 265
`errno` ... 452, 453
error handling ... 291
`error-signal` ... 302
evaluate expression, command-line argument ... 35
exception handling ... 291
exceptions ... 578
`exec_prefix` ... 413
export ... 368
expression sequencing ... 280
`extensiondir` ... 58
extensiondir ... 387

F

ffi ... 382
file descriptor ... 453
file locking ... 459
file options ... 320
File port ... 314
file system ... 460
file system combinator ... 645
file system traversal ... 644
file tree walk ... 644
finalization ... 234
finalizer ... 234
fine-grain parallelism ... 409
fluids ... 405
foreign function interface ... 382
formatted output ... 633
futures ... 409

G

GC-managed memory ... 359
Geiser ... 54
general cond clause ... 282
GNU triplet ... 414
GPL ... 1
group file ... 466
guardians, testing for GC'd objects ... 275
guild ... 54
Guile threads ... 398
`guile-2` SRFI-0 feature ... 534
guile-snarf deprecated macros ... 102
guile-snarf example ... 83
guile-snarf invocation ... 83
guile-snarf recognized macros ... 102
guile-tools ... 54
`GUILE_HISTORY` ... 629
`GUILE_LOAD_PATH` ... 353
`guileversion` ... 413

H

hash-comma ... 560
higher-order functions ... 244
host name ... 501
host name lookup ... 485
HTTP ... 502

I

i18n ... 420
IEEE-754 floating point numbers ... 167
if ... 281
iff ... 6
`includedir` ... 413
`infodir` ... 413
information encapsulation ... 365
init file, not loading ... 37
initialization ... 38
Initializing Guile ... 101
inlining ... 247
instance ... 686
integers as bits ... 586
internationalization ... 420

interpreter ... 738
invocation ... 35
invocation (command-line arguments) ... 35
IPv4 ... 484
IPv6 ... 485
iteration ... 283

J

JACAL ... 452
Jaffer, Aubrey ... 452
julian day ... 568, 571

K

keyword objects ... 590

L

lambda ... 236
`LANG` ... 501
leap second ... 568
LGPL ... 1
`libdir` ... 413
`libexecdir` ... 413
`libguileinterface` ... 413
`LIBS` ... 413
license ... 1
Line buffered input ... 653
Line continuation ... 652
Line input/output ... 310
list ... 535
list constructor ... 535
list delete ... 544
list filter ... 541
list fold ... 538
list map ... 538
list partition ... 541
list predicate ... 535
list search ... 542
list selector ... 536
list set operation ... 545
load ... 355
load path ... 54
loading srfi modules (command line) ... 37
local bindings ... 277
local environment ... 27
local time ... 469
local variable ... 27
local variables ... 277
locale ... 138, 421, 501
locale category ... 421
locale object ... 421
`localstatedir` ... 413
location ... 26
looping ... 283
low-level locale information ... 424

M

macro expansion ... 248
macros ... 248
`mandir` ... 413
match structures ... 340
math – symbolic ... 452
`memory-allocation-error` ... 302
`misc-error` ... 302
modified julian day ... 568, 571
module version ... 369
modules ... 364
multiline comments ... 345
multiple values ... 290
multiple values and cond ... 282
mutex ... 400

N

name space ... 365
name space - private ... 365
named let ... 283, 284
network ... 484
network address ... 484
network database ... 485, 488, 489
network examples ... 499
network protocols ... 490
network services ... 491
network socket ... 493
network socket address ... 492
non-local exit ... 285
`numerical-overflow` ... 302

O

object array ... 742
object table ... 742
options (command line) ... 35
options - debug ... 438
options - print ... 347
options - read ... 346
`out-of-range` ... 302
overflow, stack ... 439
overriding binding ... 368, 369

P

parallel forms ... 410
parallelism ... 409
parameter object ... 408
parameter specialize ... 575
parameter specializers ... 696
Paredit ... 53
partial evaluator ... 247
password ... 502
password file ... 466
pattern matching ... 626
pattern matching (SXML) ... 655
pattern variable ... 626

pipe ... 456, 482
pkg-config ... 57
`pkgdatadir` ... 413
`pkgincludedir` ... 413
`pkglibdir` ... 413
polar form ... 112, 118
Port ... 304
port buffering ... 458
port encoding ... 356
Port implementation ... 334
Port, block read/write ... 312
Port, C interface ... 332
Port, close ... 309
Port, default ... 313
Port, file ... 314
Port, line input/output ... 310
Port, random access ... 310
Port, soft ... 318
Port, string ... 317
Port, types ... 314
Port, void ... 319
portability between 2.0 and older versions ... 534
POSIX ... 452
POSIX threads ... 398
precise marking ... 234
`prefix` ... 413
pretty printing ... 632
primitive procedures ... 237
primitive-load ... 355
primitives ... 237
print options ... 347
procedure documentation ... 246
procedure inlining ... 247
procedure properties ... 245
procedure with setter ... 246
process group ... 482
process priority ... 478
process time ... 569
processes ... 473
Profiling ... 440
program arguments ... 471, 472
promises ... 356
prompts ... 285
protocols ... 490
ptob ... 332
pure module ... 370

Q

`q-empty` ... 650
queues ... 649

R

R6RS ... 162, 319, 591
R6RS block comments ... 345
R6RS ports ... 319
Random access, ports ... 310
re-export ... 368
read ... 356
read options ... 346
Reading ... 306
readline ... 628
readline options ... 629
receive ... 290
record ... 558
recursion ... 24
recursive expression ... 578
regex ... 335
regular expressions ... 335
`regular-expression-syntax` ... 302
remembering ... 78
replace ... 368
replacing binding ... 368, 369
reserved code point ... 129

S

sameness ... 265
`sbindir` ... 413
Scheme Shell ... 661
SCM data type ... 100
script mode ... 35
SCSH ... 661
search and replace ... 339
sequencing ... 280
service name lookup ... 485
services ... 491
setter ... 246
Setting breakpoints ... 447
Setting tracepoints ... 447
shadowing an imported variable binding ... 27
`sharedstatedir` ... 413
shell ... 38
signal ... 479
site ... 54
site path ... 54
`sitedir` ... 57
SLIB ... 451
slot ... 685
smob ... 233
socket ... 493
socket address ... 492
socket client example ... 499
socket examples ... 499
socket server example ... 500
Soft port ... 318
sorting ... 268
sorting lists ... 268
sorting vectors ... 268
source file encoding ... 355
source properties ... 432
specialize parameter ... 575
splicing ... 280
`srcdir` ... 413
SRFI ... 533

SRFI-0 533
SRFI-1 535
SRFI-10 560
SRFI-11 562
SRFI-13 562
SRFI-14 562
SRFI-16 243, 562
SRFI-17 562
SRFI-18 563
SRFI-19 568
SRFI-2 548
SRFI-23 575
SRFI-26 575
SRFI-27 576, 577
SRFI-30 578
SRFI-30 block comments 345
SRFI-31 578
SRFI-34 578
SRFI-35 578
SRFI-37 581
SRFI-38 582
SRFI-39 408, 409, 583
SRFI-4 549
SRFI-42 584
SRFI-45 584
SRFI-55 585
SRFI-6 558
SRFI-60 586
SRFI-61 282
SRFI-67 587
SRFI-69 588
SRFI-8 558
SRFI-88 590
SRFI-88 keyword syntax 180
SRFI-9 558
SRFI-98 591
stack overflow 439
`stack-overflow` 302
standard error output 314
standard input 313
standard output 313
startup (command-line arguments) 35
streams 650
String port 317
string to date 574
string, from date 572
switches (command line) 35
SXML pattern matching 655
symbolic math 452
`sysconfdir` 413
system asyncs 397
system clock 568
system name 500
`system-error` 302

T

TAI 568
tail calls 24
temporary file 465
terminal 482
textual port 321
thread time 569
threads 398
time 468, 568
time conversion 571
time formatting 470
time parsing 470
top level environment 26
`top_srcdir` 413
Trace 440
Tracepoints 447
Tracing 440
transcoder 321
transformation 248
Traps 440
truncated printing 632
Types of ports 314

U

Unicode code point 129
Unicode string encoding 167
universal time 568
unless 281
user asyncs 397
user information 466
UTC 568

V

variable arity 243, 562
variable definition 275
variable, local 27
vcell 26
VHash 226
vlist 210
VList-based hash lists 226
VM hooks 440
VM trace level 441
Void port 319

W

warnings, compiler 351
Web 502
when 281
wizards 54
word order 162
wrapped pointer types 391
Writing 308
`wrong-number-of-args` 302
`wrong-type-arg` 302
WWW 502

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 ... 689
#:allocation ... 689
#:class ... 689
#:each-subclass ... 690
#:getter ... 689
#:init-form ... 687
#:init-keyword ... 687
#:init-thunk ... 687
#:init-value ... 687
#:instance ... 689
#:metaclass ... 686
#:name ... 686
#:setter ... 689
#:slot-ref ... 690, 691
#:slot-set! ... 690, 691
#:virtual ... 690

%

% ... 287
%char-set-dump ... 136
%default-port-conversion-strategy ... 306
%library-dir ... 413
%make-void-port ... 319
%package-data-dir ... 413
%read-delimited! ... 312
%read-line ... 312
%search-load-path ... 354
%site-dir ... 54, 413
%string-dump ... 161

&

&assertion ... 610
&condition ... 608
&error ... 609
&i/o ... 611
&i/o-decoding ... 322
&i/o-encoding ... 322
&i/o-file-already-exists ... 612
&i/o-file-does-not-exist ... 612
&i/o-file-is-read-only ... 611
&i/o-file-protection ... 611
&i/o-filename ... 611
&i/o-invalid-position ... 611
&i/o-port ... 612
&i/o-read ... 611
&i/o-write ... 611
&implementation-restriction ... 610
&irritants ... 610
&lexical ... 610
&message ... 609
&no-infinities ... 618
&no-nans ... 618
&non-continuable ... 610
&serious ... 609
&syntax ... 610
&undefined ... 611
&violation ... 610
&warning ... 609
&who ... 610

'

' ... 344

(

(const ... 174
(oop goops) ... 685

*

* ... 119, 597
*scm_to_latin1_stringn ... 160
*scm_to_stringn ... 160
*scm_to_utf32_stringn ... 160
*scm_to_utf8_stringn ... 160

+

+ ... 119, 597

,

, ... 344
,@ ... 344

-

- ... 119, 597
->char-set ... 136

/

/ 119, 597

<

< 117, 596
<= 117, 596

=

= 117, 596
== 266

>

> 117, 596
>= 117, 596

@

@ 367
@@ 367

‘

‘ 344

1

1+ 119
1- 119

A

abandoned-mutex-exception? 567
abort 287, 754
abort-to-prompt 286
abs 119, 597
accept 497
access? 460
acons 222
acos 124, 596
acosh 124
activate-readline 630
adapt-response-version 520
add 756
add-duration 570
add-duration! 570
add-ephemeral-stepping-trap! 448
add-ephemeral-trap-at-frame-finish! 448
add-hook! 272
add-method! 719
add-ref-resolver! 679
add-to-load-path 354
add-trace-at-procedure-call! 447
add-trap! 446
add-trap-at-procedure-call! 447
add-trap-at-source-location! 448
add1 756
addrinfo:addr 488
addrinfo:canonname 488
addrinfo:fam 487
addrinfo:flags 487
addrinfo:protocol 487
addrinfo:socktype 487
alarm 480
alignof 392
alist->hash-table 589
alist->vhash 228
alist-cons 545
alist-copy 545
alist-delete 545
alist-delete! 545
all-threads 399
and 282, 595
and-let* 548
angle 118, 595
any 543
any->c32vector 558
any->c64vector 558
any->f32vector 558
any->f64vector 558
any->s16vector 558
any->s32vector 558
any->s64vector 558
any->s8vector 558
any->u16vector 558
any->u32vector 558
any->u64vector 558
any->u8vector 558
any-bits-set? 586
append 187, 597
append! 187
append-map 541
append-map! 541
append-reverse 538
append-reverse! 538
apply 349, 600, 746
apply-templates 667
apply:nconc2last 349
apropos 47
apropos-completion-function 631
args-fold 582
arithmetic-shift 586
arity:allow-other-keys? 239
arity:end 239
arity:kw 239
arity:nopt 239
arity:nreq 239
arity:rest? 239
arity:start 239
array->list 202
array-contents 205
array-copy! 202
array-copy-in-order! 202
array-dimensions 201

array-equal? ... 202
array-fill! ... 202
array-for-each ... 202
array-in-bounds? ... 201
array-index-map! ... 202
array-map! ... 202
array-map-in-order! ... 202
array-rank ... 201
array-ref ... 201
array-set! ... 201
array-shape ... 201
array-type ... 201
array? ... 200
ash ... 125, 756
asin ... 124, 596
asinh ... 124
assert ... 599
assert-curr-char ... 672
assert-nargs-ee ... 748
assert-nargs-ee/locals ... 749
assert-nargs-ge ... 748
assert-nargs-ge/locals ... 749
assertion-violation ... 599
assertion-violation? ... 610
assoc ... 222, 544, 602
assoc-ref ... 222
assoc-remove! ... 224
assoc-set! ... 222
assp ... 602
assq ... 222, 602
assq-ref ... 222
assq-remove! ... 224
assq-set! ... 222
assv ... 222, 602
assv-ref ... 222
assv-remove! ... 224
assv-set! ... 222
async ... 398
async-mark ... 398
atan ... 124, 596
atanh ... 124
attlist->alist ... 671
attlist-add ... 671
attlist-fold ... 671
attlist-null? ... 671
attlist-remove-top ... 671

B

backtrace ... 49, 437
basename ... 466
begin ... 280, 595
begin-thread ... 400
binary-port? ... 324
bind ... 496
bind-kwargs ... 749
bind-optionals ... 748
bind-optionals/shuffle ... 749
bind-rest ... 748
bind-textdomain-codeset ... 429
binding ... 48
binding:boxed? ... 239
binding:end ... 239
binding:index ... 239
binding:name ... 239
binding:start ... 239
bindtextdomain ... 428
bit-count ... 196, 586
bit-count* ... 196
bit-extract ... 127
bit-field ... 586
bit-invert! ... 196
bit-position ... 196
bit-set*! ... 196
bit-set? ... 586
bitvector ... 195
bitvector->list ... 195
bitvector-fill! ... 195
bitvector-length ... 195
bitvector-ref ... 195
bitvector-set! ... 195
bitvector? ... 195
bitwise-and ... 586, 619
bitwise-arithmetic-shift ... 620
bitwise-arithmetic-shift-left ... 620
bitwise-arithmetic-shift-right ... 620
bitwise-bit-count ... 619
bitwise-bit-field ... 619
bitwise-bit-set? ... 619
bitwise-copy-bit ... 619
bitwise-copy-bit-field ... 619
bitwise-first-bit-set ... 619
bitwise-if ... 586, 619
bitwise-ior ... 586, 619
bitwise-length ... 619
bitwise-merge ... 586
bitwise-not ... 586, 619
bitwise-reverse-bit-field ... 620
bitwise-rotate-bit-field ... 620
bitwise-xor ... 586, 619
boolean? ... 105, 593
booleans->integer ... 587
bound-identifier=? ... 258, 620
box ... 744
br ... 750
br-if ... 750
br-if-eq ... 751
br-if-nargs-gt ... 748
br-if-nargs-lt ... 748
br-if-nargs-ne ... 748
br-if-not ... 750
br-if-not-eq ... 751
br-if-not-null ... 751
br-if-null ... 751
break ... 50, 284, 542, 755
break! ... 542

break-at-source ... 50
broadcast-condition-variable ... 403
buffer-mode ... 321
buffer-mode? ... 321
build-request ... 518
build-response ... 520
build-uri ... 505
bv-f32-native-ref ... 757
bv-f32-native-set ... 757
bv-f32-ref ... 757
bv-f32-set ... 757
bv-f64-native-ref ... 757
bv-f64-native-set ... 757
bv-f64-ref ... 757
bv-f64-set ... 757
bv-s16-native-ref ... 757
bv-s16-native-set ... 757
bv-s16-ref ... 757
bv-s16-set ... 757
bv-s32-native-ref ... 757
bv-s32-native-set ... 757
bv-s32-ref ... 757
bv-s32-set ... 757
bv-s64-native-ref ... 757
bv-s64-native-set ... 757
bv-s64-ref ... 757
bv-s64-set ... 757
bv-s8-ref ... 757
bv-s8-set ... 757
bv-u16-native-ref ... 757
bv-u16-native-set ... 757
bv-u16-ref ... 757
bv-u16-set ... 757
bv-u32-native-ref ... 757
bv-u32-native-set ... 757
bv-u32-ref ... 757
bv-u32-set ... 757
bv-u64-native-ref ... 757
bv-u64-native-set ... 757
bv-u64-ref ... 757
bv-u64-set ... 757
bv-u8-ref ... 757
bv-u8-set ... 757
bytecode->objcode ... 769
bytevector->pointer ... 390
bytevector->sint-list ... 166
bytevector->string ... 324
bytevector->u8-list ... 166
bytevector->uint-list ... 166
bytevector-copy ... 163
bytevector-copy! ... 163
bytevector-fill! ... 163
bytevector-ieee-double-native-ref ... 167
bytevector-ieee-double-native-set! ... 167
bytevector-ieee-double-ref ... 167
bytevector-ieee-double-set! ... 167
bytevector-ieee-single-native-ref ... 167
bytevector-ieee-single-native-set! ... 167
bytevector-ieee-single-ref ... 167
bytevector-ieee-single-set! ... 167
bytevector-length ... 163
bytevector-s16-native-ref ... 165
bytevector-s16-native-set! ... 166
bytevector-s16-ref ... 164
bytevector-s16-set! ... 165
bytevector-s32-native-ref ... 165
bytevector-s32-native-set! ... 166
bytevector-s32-ref ... 165
bytevector-s32-set! ... 165
bytevector-s64-native-ref ... 165
bytevector-s64-native-set! ... 166
bytevector-s64-ref ... 165
bytevector-s64-set! ... 165
bytevector-s8-ref ... 164
bytevector-s8-set! ... 165
bytevector-sint-ref ... 164
bytevector-sint-set! ... 164
bytevector-u16-native-ref ... 165
bytevector-u16-native-set! ... 166
bytevector-u16-ref ... 164
bytevector-u16-set! ... 165
bytevector-u32-native-ref ... 165
bytevector-u32-native-set! ... 166
bytevector-u32-ref ... 165
bytevector-u32-set! ... 165
bytevector-u64-native-ref ... 165
bytevector-u64-native-set! ... 166
bytevector-u64-ref ... 165
bytevector-u64-set! ... 165
bytevector-u8-ref ... 164
bytevector-u8-set! ... 165
bytevector-uint-ref ... 164
bytevector-uint-set! ... 164
bytevector=? ... 163
bytevector? ... 163

C

c32vector ... 551
c32vector->list ... 554
c32vector-length ... 552
c32vector-ref ... 552
c32vector-set! ... 553
c32vector? ... 550
c64vector ... 551
c64vector->list ... 554
c64vector-length ... 552
c64vector-ref ... 552
c64vector-set! ... 553
c64vector? ... 550
caaaar ... 183, 594
caaadr ... 183, 594
caaar ... 183, 593
caadar ... 183, 594
caaddr ... 183, 594
caadr ... 183, 593

caar ... 183, 593
cadaar ... 183, 594
cadadr ... 183, 594
cadar ... 183, 593
caddar ... 183, 594
cadddr ... 183, 594
caddr ... 183, 593
cadr ... 183, 593
call ... 746
call-with-blocked-asyncs ... 397
call-with-current-continuation ... 288, 599
call-with-error-handling ... 438
call-with-file-and-dir ... 677
call-with-input-file ... 316, 612
call-with-input-string ... 317
call-with-new-thread ... 399
call-with-output-file ... 316, 612
call-with-output-string ... 317
call-with-port ... 325
call-with-prompt ... 285
call-with-trace ... 446
call-with-unblocked-asyncs ... 398
call-with-values ... 291, 599
call/cc ... 288, 599, 748
call/nargs ... 747
cancel-thread ... 400
car ... 183, 593, 755
car+cdr ... 537
case ... 282, 595
case-lambda ... 243, 603
case-lambda* ... 244
catch ... 293
cd ... 473
cdaaar ... 183, 594
cdaadr ... 183, 594
cdaar ... 183, 593
cdadar ... 183, 594
cdaddr ... 183, 594
cdadr ... 183, 594
cdar ... 183, 593
cddaar ... 183, 594
cddadr ... 183, 594
cddar ... 183, 594
cdddar ... 183, 594
cddddr ... 183, 594
cdddr ... 183, 594
cddr ... 183, 593
cdr ... 183, 593, 755
ceiling ... 120, 597
ceiling-quotient ... 121
ceiling-remainder ... 121
ceiling/ ... 121
center-string ... 680
centered-quotient ... 122
centered-remainder ... 122
centered/ ... 122
change-class ... 723
char->integer ... 132, 593
char-alphabetic? ... 131, 600
char-ci<=? ... 131, 600
char-ci<? ... 131, 600
char-ci=? ... 131, 600
char-ci>=? ... 131, 600
char-ci>? ... 131, 600
char-downcase ... 132, 600
char-foldcase ... 600
char-general-category ... 132, 600
char-is-both? ... 131
char-locale-ci<? ... 422
char-locale-ci=? ... 422
char-locale-ci>? ... 422
char-locale-downcase ... 423
char-locale-titlecase ... 423
char-locale-upcase ... 423
char-locale<? ... 422
char-locale>? ... 422
char-lower-case? ... 131, 600
char-numeric? ... 131, 600
char-ready? ... 306
char-set ... 135
char-set->list ... 136
char-set->string ... 136
char-set-adjoin ... 137
char-set-adjoin! ... 137
char-set-any ... 137
char-set-complement ... 137
char-set-complement! ... 138
char-set-contains? ... 137
char-set-copy ... 135
char-set-count ... 136
char-set-cursor ... 133
char-set-cursor-next ... 134
char-set-delete ... 137
char-set-delete! ... 137
char-set-diff+intersection ... 138
char-set-diff+intersection! ... 138
char-set-difference ... 138
char-set-difference! ... 138
char-set-every ... 137
char-set-filter ... 135
char-set-filter! ... 135
char-set-fold ... 134
char-set-for-each ... 134
char-set-hash ... 133
char-set-intersection ... 138
char-set-intersection! ... 138
char-set-map ... 134
char-set-ref ... 134
char-set-size ... 136
char-set-unfold ... 134
char-set-unfold! ... 134
char-set-union ... 137
char-set-union! ... 138
char-set-xor ... 138
char-set-xor! ... 138
char-set<= ... 133

char-set= ... 133
char-set? ... 133
char-title-case? ... 600
char-titlecase ... 132, 600
char-upcase ... 132, 600
char-upper-case? ... 131, 600
char-whitespace? ... 131, 600
char<=? ... 130, 593
char<? ... 130, 593
char=? ... 130, 593
char>=? ... 130, 593
char>? ... 130, 593
char? ... 130, 593
chdir ... 473
chmod ... 463
chown ... 463
chroot ... 473
circular-list ... 535
circular-list? ... 536
class ... 714
class-direct-methods ... 702
class-direct-slots ... 702
class-direct-subclasses ... 702
class-direct-supers ... 702
class-methods ... 703
class-name ... 702
class-of ... 703, 756
class-precedence-list ... 703
class-redefinition ... 722
class-slot-definition ... 703
class-slot-ref ... 707
class-slot-set! ... 707
class-slots ... 703
class-subclasses ... 703
clear-value-history! ... 47
close ... 456
close-fdes ... 456
close-input-port ... 309, 613
close-output-port ... 309, 613
close-pipe ... 483
close-port ... 309
close-server ... 524
closedir ... 464
collapse-repeated-chars ... 681
command-line ... 471, 613
compile ... 48, 351
compile-file ... 48, 351
compiled-file-name ... 352
complex? ... 113, 595
compose ... 245
compute-std-cpl ... 717
concatenate ... 537
concatenate! ... 537
cond ... 281, 595
cond-expand ... 533
condition ... 580, 609
condition-accessor ... 609
condition-has-type? ... 579
condition-irritants ... 610
condition-message ... 580, 609
condition-predicate ... 609
condition-ref ... 579
condition-type? ... 579
condition-variable-broadcast! ... 566
condition-variable-name ... 566
condition-variable-signal! ... 566
condition-variable-specific ... 566
condition-variable-specific-set! ... 566
condition-variable? ... 403, 566
condition-who ... 610
condition? ... 608
connect ... 496
cons ... 182, 593, 755
cons* ... 186, 602
cons-source ... 433
const ... 244
continuation-call ... 750
continue ... 284
copy-bit ... 587
copy-bit-field ... 587
copy-file ... 463
copy-random-state ... 127
copy-time ... 569
copy-tree ... 270
cos ... 123, 596
cosh ... 124
count ... 538
coverage-data->lcov ... 448
coverage-data? ... 448
crypt ... 501
ctermid ... 482
current-date ... 571
current-dynamic-state ... 407
current-error-port ... 314, 330
current-exception-handler ... 567
current-filename ... 433
current-input-port ... 313, 326
current-julian-day ... 571
current-load-port ... 353
current-modified-julian-day ... 571
current-module ... 375
current-output-port ... 313, 330
current-processor-count ... 479
current-source-location ... 433
current-ssax-error-port ... 671
current-thread ... 399, 563
current-time ... 468, 566, 569
cut ... 575
cute ... 575

D

date->julian-day ... 571
date->modified-julian-day ... 571
date->string ... 572
date->time-monotonic ... 571

date->time-tai ... 571
date->time-utc ... 571
date-day ... 570
date-hour ... 570
date-minute ... 570
date-month ... 571
date-nanosecond ... 570
date-second ... 570
date-week-day ... 571
date-week-number ... 571
date-year ... 571
date-year-day ... 571
date-zone-offset ... 571
date? ... 570
datum->random-state ... 128
datum->syntax ... 256, 621
debug-disable ... 439
debug-enable ... 439
debug-options ... 438
debug-set! ... 439
declare-default-port! ... 505
declare-header! ... 507
declare-opaque-header! ... 507
deep-clone ... 708
default-duplicate-binding-handler ... 370
default-prompt-tag ... 285
default-random-source ... 576
define ... 276, 594, 745
define* ... 240
define*-public ... 243
define-accessor ... 694
define-class ... 685
define-condition-type ... 580, 609
define-enumeration ... 624
define-generic ... 693
define-inlinable ... 247
define-language ... 758
define-macro ... 260
define-method ... 693
define-module ... 367
define-once ... 276
define-parsed-entity! ... 671
define-public ... 370
define-reader-ctor ... 560
define-record-type ... 559, 604
define-server-impl ... 523
define-syntax ... 248, 594
define-syntax-parameter ... 262
define-syntax-rule ... 253
define-wrapped-pointer-type ... 391
defined? ... 279
defmacro ... 260
defmacro* ... 243
defmacro*-public ... 243
defvar ... 276
delay ... 356, 584, 625
delete ... 188, 544
delete! ... 188, 544
delete-duplicates ... 544
delete-duplicates! ... 544
delete-file ... 463
delete-trap! ... 447
delete1! ... 189
delq ... 188
delq! ... 188
delq1! ... 189
delv ... 188
delv! ... 188
delv1! ... 189
denominator ... 112, 596
deq! ... 649
dereference-pointer ... 390
describe ... 47
directory-stream? ... 464
dirname ... 466
disable-trap! ... 447
disable-value-history! ... 46
disassemble ... 48
disassemble-file ... 48
display ... 347, 613, 709
display-application ... 432
display-backtrace ... 431
display-error ... 302
div ... 598, 756
div-and-mod ... 598
div0 ... 598
div0-and-mod0 ... 598
do ... 283, 603
dotted-list? ... 536
doubly-weak-hash-table? ... 363
down ... 49
drain-input ... 308
drop ... 537, 755
drop-right ... 537
drop-right! ... 537
drop-while ... 542
dup ... 457, 755
dup->fdes ... 457
dup->inport ... 457
dup->outport ... 457
dup->port ... 457
dup2 ... 458
duplicate-port ... 457
dynamic-call ... 384
dynamic-func ... 384
dynamic-link ... 383
dynamic-object? ... 383
dynamic-pointer ... 389
dynamic-state? ... 407
dynamic-unlink ... 383
dynamic-wind ... 299, 600

E

eager ... 584
ee? ... 756

effective-version 412
eighth 536
empty-box 744
enable-primitive-generic! 694
enable-trap! 446
enable-value-history! 46
encode-and-join-uri-path 506
end-of-char-set? 134
endgrent 468
endhostent 489
endianness 162
endnetent 490
endprotoent 491
endpwent 467
endservent 492
enq! 649
ensure-accessor 720
ensure-generic 719
ensure-metaclass 715
enum-set->list 623
enum-set-complement 624
enum-set-constructor 623
enum-set-difference 624
enum-set-indexer 623
enum-set-intersection 624
enum-set-member? 624
enum-set-projection 624
enum-set-subset? 624
enum-set-union 624
enum-set-universe 623
enum-set=? 624
environ 473
environment 625
eof-object 324, 612
eof-object? 306, 324, 612
eol-style 321
eq? 265, 595, 755
equal-hash 623
equal? 266, 595, 755
eqv? 266, 595, 755
error 49, 297, 599
error-handling-mode 323
error-message 49
error? 581, 609
escape-special-chars 679
euclidean-quotient 120
euclidean-remainder 120
euclidean/ 120
eval 348, 624
eval-string 348
eval-when 264, 353
even? 115, 596
every 543
exact 596
exact->inexact 114, 625
exact-integer-sqrt 116, 596
exact? 113, 596
execl 477
execle 477
execlp 477
exists 601
exit 614
exp 124, 595
expand 48
expand-tabs 680
expect 655
expect-strings 653
export 370
export! 370
expt 123, 595
extract-condition 579

F

f32vector 551
f32vector->list 554
f32vector-length 552
f32vector-ref 552
f32vector-set! 553
f32vector? 550
f64vector 551
f64vector->list 554
f64vector-length 552
f64vector-ref 552
f64vector-set! 553
f64vector? 550
false-if-exception 298
fchmod 463
fchown 463
fcntl 458
fdes->inport 455
fdes->outport 455
fdes->ports 455
fdopen 454
feature? 414
fflush 309
fifth 536
file-encoding 356
file-exists? 466
file-options 320
file-port? 316
file-system-fold 645
file-system-tree 644
filename-completion-function 631
fileno 454
fill-string 682
filter 189, 601, 675
filter! 189
filter-empty-elements 678
filter-map 541
find 542, 601
find-string-from-port? 673
find-tail 542
finish 50
finite? 112, 599
first 536

first-set-bit ... 586
fix-closure ... 744
fixnum->flonum ... 619
fixnum-width ... 614
fixnum? ... 614
fl* ... 617
fl+ ... 617
fl- ... 617
fl/ ... 617
fl<=? ... 617
fl<? ... 617
fl=? ... 617
fl>=? ... 617
fl>? ... 617
flabs ... 617
flacos ... 618
flasin ... 618
flatan ... 618
flceiling ... 618
flcos ... 618
fldenominator ... 618
fldiv ... 617
fldiv-and-mod ... 617
fldiv0 ... 617
fldiv0-and-mod0 ... 617
fldmod ... 617
fleven? ... 617
flexp ... 618
flexpt ... 618
flfinite? ... 617
flfloor ... 618
flinfinite? ... 617
flinteger? ... 617
fllog ... 618
flmax ... 617
flmin ... 617
flmod0 ... 617
flnan? ... 617
flnegative? ... 617
flnumerator ... 618
flock ... 459
flodd? ... 617
flonum? ... 616
floor ... 120, 597
floor-quotient ... 120
floor-remainder ... 120
floor/ ... 120
flpositive? ... 617
flround ... 618
flsin ... 618
flsqrt ... 618
fltan ... 618
fltruncate ... 618
fluid-bound? ... 406
fluid-ref ... 406, 754
fluid-set ... 754
fluid-set! ... 406
fluid-unset! ... 406
fluid? ... 406
flush-all-ports ... 309
flush-output-port ... 330
flzero? ... 617
fold ... 538
fold-layout ... 667
fold-left ... 601
fold-matches ... 338
fold-right ... 538, 601
fold-values ... 667
foldt ... 667
foldts ... 667, 674
foldts* ... 667
foldts*-values ... 667
for-all ... 601
for-each ... 190, 541, 596
force ... 356, 584, 625
force-output ... 309
foreign-call ... 750
format ... 633
fourth ... 536
frame ... 49
frame-address ... 432
frame-arguments ... 431
frame-dynamic-link ... 432
frame-instruction-pointer ... 432
frame-local-ref ... 432
frame-local-set! ... 432
frame-mv-return-address ... 432
frame-num-locals ... 432
frame-previous ... 431
frame-procedure ... 431
frame-return-address ... 432
frame-stack-pointer ... 432
frame? ... 431
free-boxed-ref ... 744
free-boxed-set ... 744
free-identifier=? ... 258, 620
free-ref ... 744
fstat ... 461
fsync ... 455
ftell ... 310
ftruncate ... 310
ftw ... 647
future ... 410
future? ... 410
fx* ... 614
fx*/carry ... 615
fx+ ... 614
fx+/carry ... 615
fx- ... 615
fx-/carry ... 615
fx<=? ... 614
fx<? ... 614
fx=? ... 614
fx>=? ... 614
fx>? ... 614
fxand ... 615

fxarithmetic-shift 616
fxarithmetic-shift-left 616
fxarithmetic-shift-right 616
fxbit-count 616
fxbit-field 616
fxbit-set? 616
fxcopy-bit 616
fxcopy-bit-field 616
fxdiv 615
fxdiv-and-mod 615
fxdiv0 615
fxdiv0-and-mod0 615
fxeven? 614
fxfirst-bit-set 616
fxif 615
fxior 615
fxlength 616
fxmax 614
fxmin 614
fxmod 615
fxmod0 615
fxnegative? 614
fxnot 615
fxodd? 614
fxpositive? 614
fxreverse-bit-field 616
fxrotate-bit-field 616
fxxor 615
fxzero? 614

G

gc 50, 358
gc-live-object-stats 359
gc-stats 359
gcd 116, 596
gcprof 666
ge? 756
generalized-vector->list 197
generalized-vector-length 197
generalized-vector-ref 197
generalized-vector-set! 197
generalized-vector? 197
generate-temporaries 258, 621
generic-function-methods 705
generic-function-name 705
gensym 174
get-bytevector-all 328
get-bytevector-n 328
get-bytevector-n! 328
get-bytevector-some 328
get-char 328
get-datum 329
get-environment-variable 591
get-environment-variables 591
get-internal-real-time 471
get-internal-run-time 471
get-line 329
get-output-string 318
get-print-state 308
get-string-all 329
get-string-n 328
get-string-n! 329
get-u8 328
getaddrinfo 485
getaffinity 478
getcwd 473
getegid 474
getenv 472
geteuid 474
getgid 474
getgr 468
getgrent 468
getgrgid 467
getgrnam 467
getgroups 473
gethost 488
gethostbyaddr 488
gethostbyname 488
gethostent 489
gethostname 501
getitimer 481
getlogin 468
getnet 489
getnetbyaddr 489
getnetbyname 489
getnetent 490
getopt-long 531
getpass 502
getpeername 497
getpgrp 475
getpid 473
getppid 474
getpriority 478
getproto 490
getprotobyname 490
getprotobynumber 490
getprotoent 491
getpw 467
getpwent 467
getpwnam 467
getpwuid 467
getserv 491
getservbyname 491
getservbyport 491
getservent 492
getsid 475
getsockname 497
getsockopt 494
getter-with-setter 563
gettext 427
gettimeofday 468
getuid 474
gmtime 469
goops-error 708
greatest-fixnum 614

group:gid ... 467
group:mem ... 467
group:name ... 467
group:passwd ... 467
gt? ... 756
guard ... 607
guild compile ... 350
GUILE_CHECK_RETVAL ... 96
GUILE_FLAGS ... 95
GUILE_MODULE_AVAILABLE ... 96
GUILE_MODULE_CHECK ... 96
GUILE_MODULE_EXPORTS ... 96
GUILE_MODULE_REQUIRED ... 96
GUILE_MODULE_REQUIRED_EXPORT ... 96
GUILE_PROGS ... 95
GUILE_SITE_DIR ... 96

H

halt ... 755
handle-request ... 523
hash ... 232, 590
hash-by-identity ... 590
hash-clear! ... 231
hash-create-handle! ... 232
hash-fold ... 233
hash-for-each ... 232
hash-for-each-handle ... 233
hash-get-handle ... 232
hash-map->list ... 232
hash-ref ... 231
hash-remove! ... 231
hash-set! ... 231
hash-table->alist ... 590
hash-table-delete! ... 589
hash-table-equivalence-function ... 590
hash-table-exists? ... 589
hash-table-fold ... 590
hash-table-hash-function ... 590
hash-table-keys ... 589
hash-table-ref ... 589
hash-table-ref/default ... 589
hash-table-set! ... 589
hash-table-size ... 589
hash-table-update! ... 589
hash-table-update!/default ... 589
hash-table-values ... 589
hash-table-walk ... 589
hash-table? ... 231
hashq ... 232
hashq-create-handle! ... 232
hashq-get-handle ... 232
hashq-ref ... 231
hashq-remove! ... 231
hashq-set! ... 231
hashtable-clear! ... 622
hashtable-contains? ... 622
hashtable-copy ... 622
hashtable-delete! ... 622
hashtable-entries ... 622
hashtable-equivalence-function ... 622
hashtable-hash-function ... 623
hashtable-keys ... 622
hashtable-mutable? ... 623
hashtable-ref ... 622
hashtable-set! ... 622
hashtable-size ... 622
hashtable-update! ... 622
hashtable? ... 622
hashv ... 232
hashv-create-handle! ... 232
hashv-get-handle ... 232
hashv-ref ... 231
hashv-remove! ... 231
hashv-set! ... 231
hashx-create-handle! ... 232
hashx-get-handle ... 232
hashx-ref ... 231
hashx-remove! ... 231
hashx-set! ... 231
header->string ... 506
header-parser ... 507
header-validator ... 507
header-writer ... 507
help ... 47
hook->list ... 272
hook-empty? ... 272
hook? ... 272
hostent:addr-list ... 488
hostent:addrtype ... 488
hostent:aliases ... 488
hostent:length ... 488
hostent:name ... 488
htonl ... 499
htons ... 499
http ... 524
http-get ... 522

I

i/o-decoding-error? ... 322
i/o-encoding-error-char ... 322
i/o-encoding-error? ... 322
i/o-error-filename ... 611
i/o-error-port ... 612
i/o-error-position ... 611
i/o-error? ... 611
i/o-file-already-exists-error? ... 612
i/o-file-does-not-exist-error? ... 612
i/o-file-is-read-only-error? ... 611
i/o-file-protection-error? ... 611
i/o-filename-error? ... 611
i/o-invalid-position-error? ... 611
i/o-port-error? ... 612
i/o-read-error? ... 611
i/o-write-error? ... 611

identifier-syntax 261, 262, 594
identifier? 255, 620
identity 245
if 281, 595
imag-part 118, 595
implementation-restriction-violation? ... 610
import 48, 373
in 48
include 357
include-from-path 358
inet-aton 484
inet-lnaof 485
inet-makeaddr 485
inet-netof 484
inet-ntoa 484
inet-ntop 485
inet-pton 485
inexact 596
inexact->exact 114, 625
inexact? 114, 596
inf 112
inf? 111
infinite? 599
input-port? 305, 325, 612
inspect 50
install-trap-handler! 447
instance? 703
instrumented-source-files 449
instrumented/executed-lines 449
integer->char 132, 593
integer->list 587
integer-expt 126
integer-length 126
integer-valued? 598
integer? 107, 596
interaction-environment 348
iota 535
irritants-condition? 610
is-a? 703
isatty? 482

J

join-thread 399
join-timeout-exception? 567
julian-day->date 571
julian-day->time-monotonic 571
julian-day->time-tai 571
julian-day->time-utc 571

K

keyword->string 590
keyword->symbol 181
keyword? 181, 590
kill 479
known-header? 507

L

lalr-parser 343
lambda 237, 594
lambda* 240
language 48
last 537
last-pair 187
latin-1-codec 321
lazy 584
lchown 463
lcm 116, 596
le? 756
least-fixnum 614
left-justify-string 681
length 186, 597
length+ 537
let 277, 285, 595
let* 277, 595
let*-values 562, 595
let-keywords 242
let-keywords* 242
let-optional 242
let-optional* 242
let-syntax 248, 594
let-values 562, 595
letpar 411
letrec 277, 595
letrec* 278, 595
letrec-syntax 249, 594
lexical-violation? 610
library 372
line-execution-counts 449
link 464
link-now 745
list 186, 596, 752
list->array 200
list->bitvector 195
list->c32vector 554
list->c64vector 554
list->char-set 135
list->char-set! 135
list->f32vector 554
list->f64vector 554
list->integer 587
list->s16vector 554
list->s32vector 554
list->s64vector 554
list->s8vector 554
list->stream 651
list->string 143, 597
list->symbol 172
list->typed-array 201
list->u16vector 554
list->u32vector 554
list->u64vector 554
list->u8vector 554
list->vector 191, 599
list->vlist 212

list->weak-vector 363
list-cdr-ref 187
list-cdr-set! 188
list-copy 186, 535
list-head 187
list-index 543
list-matches 338
list-ref 187, 597
list-set! 188
list-sort 602
list-tabulate 535
list-tail 187, 597
list-traps 446
list= 536
list? 185, 593, 755
listen 496
load 48, 352
load-array 753
load-compiled 352
load-extension 384
load-from-path 353
load-number 753
load-objcode 769
load-program 753
load-string 753
load-symbol 753
load-wide-string 753
local-bound? 745
local-boxed-ref 744
local-boxed-set 744
local-compile 357
local-eval 357
local-ref 743
local-set 744
locale-am-string 425
locale-currency-symbol 426
locale-currency-symbol-precedes-negative? 426
locale-currency-symbol-precedes-positive? 426
locale-date+time-format 425
locale-date-format 425
locale-day 424
locale-day-short 424
locale-decimal-point 425
locale-digit-grouping 425
locale-encoding 424
locale-era 425
locale-era-date+time-format 425
locale-era-date-format 425
locale-era-time-format 425
locale-era-year 425
locale-monetary-decimal-point 425
locale-monetary-fractional-digits 426
locale-monetary-grouping 425
locale-monetary-negative-sign 426
locale-monetary-positive-sign 426
locale-monetary-thousands-separator 425
locale-month 424
locale-month-short 424
locale-negative-separated-by-space? 426
locale-negative-sign-position 426
locale-no-regexp 427
locale-pm-string 425
locale-positive-separated-by-space? 426
locale-positive-sign-position 426
locale-string->inexact 424
locale-string->integer 423
locale-thousands-separator 425
locale-time+am/pm-format 425
locale-time-format 425
locale-yes-regexp 427
locale? 421
locals 49
localtime 469
lock-mutex 401
log 124, 595
log10 124
log2-binary-factors 586
logand 124, 756
logbit? 125
logcount 126
logior 125, 756
lognot 125
logtest 125
logxor 125, 756
long-local-bound? 745
long-local-ref 743
long-local-set 744
long-object-ref 752
long-toplevel-ref 745
long-toplevel-set 745
lookahead-char 328
lookahead-u8 328
lookup-compilation-order 759
lookup-language 758
lookup-server-impl 523
lset-adjoin 546
lset-diff+intersection 547
lset-diff+intersection! 547
lset-difference 547
lset-difference! 547
lset-intersection 547
lset-intersection! 547
lset-union 546
lset-union! 546
lset-xor 547
lset-xor! 547
lset<= 545
lset= 546
lstat 462
lt? 756

M

macro-binding 265

macro-name ... 265
macro-transformer ... 265
macro-type ... 264
macro? ... 264
magnitude ... 118, 595
major-version ... 412
make ... 686, 715
make-accessor ... 720
make-arbiter ... 396
make-array ... 200, 752
make-assertion-violation ... 610
make-binding ... 239
make-bitvector ... 195
make-buffered-input-port ... 653
make-bytevector ... 163
make-c-struct ... 393
make-c32vector ... 551
make-c64vector ... 551
make-char32 ... 752
make-char8 ... 751
make-chunked-input-port ... 516
make-chunked-output-port ... 517
make-class ... 714
make-closure ... 744
make-completion-function ... 631
make-compound-condition ... 579
make-condition ... 579
make-condition-type ... 579
make-condition-variable ... 402, 566
make-custom-binary-input-port ... 327
make-custom-binary-output-port ... 331
make-date ... 570
make-doubly-weak-hash-table ... 363
make-dynamic-state ... 407
make-empty-attlist ... 671
make-enumeration ... 623
make-eol ... 751
make-eq-hashtable ... 621
make-eqv-hashtable ... 621
make-error ... 609
make-f32vector ... 551
make-f64vector ... 551
make-false ... 751
make-fluid ... 406
make-future ... 410
make-generic ... 720
make-guardian ... 364
make-hash-table ... 231, 588
make-hashtable ... 621
make-hook ... 272
make-i/o-decoding-error ... 322
make-i/o-encoding-error ... 322
make-i/o-error ... 611
make-i/o-file-already-exists-error ... 612
make-i/o-file-does-not-exist-error ... 612
make-i/o-file-is-read-only-error ... 611
make-i/o-file-protection-error ... 611
make-i/o-invalid-position-error ... 611
make-i/o-port-error ... 612
make-i/o-read-error ... 611
make-i/o-write-error ... 611
make-implementation-restriction-violation ... 610
make-instance ... 687
make-int16 ... 751
make-int64 ... 751
make-int8 ... 751
make-int8:0 ... 751
make-int8:1 ... 751
make-io-filename-error ... 611
make-irritants-condition ... 610
make-keyword ... 752
make-lexical-violation ... 610
make-line-buffered-input-port ... 653
make-list ... 186
make-locale ... 421
make-message-condition ... 609
make-method ... 719
make-mutex ... 401, 565
make-nil ... 751
make-no-infinities-violation ... 618
make-no-nans-violation ... 618
make-non-continuable-violation ... 610
make-object-property ... 267
make-parameter ... 408
make-pointer ... 389
make-polar ... 118, 595
make-procedure-with-setter ... 247
make-program ... 770
make-prompt-tag ... 285
make-q ... 649
make-random-source ... 577
make-record-constructor-descriptor ... 606
make-record-type ... 213
make-record-type-descriptor ... 605
make-rectangular ... 118, 595
make-recursive-mutex ... 401
make-regexp ... 337
make-s16vector ... 551
make-s32vector ... 551
make-s64vector ... 551
make-s8vector ... 551
make-serious-condition ... 609
make-shared-array ... 203
make-socket-address ... 492
make-soft-port ... 318
make-stack ... 430
make-stream ... 651
make-string ... 143, 597
make-struct ... 215, 752
make-struct-layout ... 218
make-symbol ... 177, 752
make-syntax-transformer ... 264
make-syntax-violation ... 610
make-text-wrapper ... 681
make-thread ... 400, 564

make-time ... 569
make-transcoder ... 323
make-true ... 751
make-typed-array ... 200
make-u16vector ... 551
make-u32vector ... 551
make-u64vector ... 551
make-u8vector ... 551
make-uint64 ... 751
make-unbound-fluid ... 406
make-undefined-variable ... 374
make-undefined-violation ... 611
make-variable ... 374, 746
make-variable-transformer ... 261, 620
make-vector ... 191, 599
make-violation ... 610
make-vtable ... 214
make-vtable-vtable ... 217
make-warning ... 609
make-weak-key-hash-table ... 363
make-weak-value-hash-table ... 363
make-weak-vector ... 363
make-who-condition ... 610
malloc-stats ... 361
map ... 190, 541
map! ... 541
map-in-order ... 190
map-union ... 675
match ... 626
match:count ... 341
match:end ... 341
match:prefix ... 341
match:start ... 340
match:string ... 341
match:substring ... 340
match:suffix ... 341
max ... 119, 597
member ... 189, 543, 602
memp ... 602
memq ... 189, 602
memv ... 189, 602
merge ... 268
merge! ... 268
message-condition? ... 580, 609
method ... 718
method-generic-function ... 705
method-procedure ... 705
method-source ... 705
method-specializers ... 705
micro-version ... 412
min ... 119, 597
minor-version ... 412
mkdir ... 464
mknod ... 465
mkstemp! ... 465
mktime ... 470
mod ... 598, 756
mod0 ... 598
modified-julian-day->date ... 571
modified-julian-day->time-monotonic ... 571
modified-julian-day->time-tai ... 571
modified-julian-day->time-utc ... 571
module ... 48
module-add! ... 376
module-define! ... 376
module-ref ... 376
module-set! ... 376
module-stexi-documentation ... 683
module-use! ... 375
module-uses ... 375
module-variable ... 376
modulo ... 116, 625
modulo-expt ... 116
monetary-amount->locale-string ... 424
monitor ... 403
move->fdes ... 455
mul ... 756
mutex-level ... 402
mutex-lock! ... 565
mutex-locked? ... 402
mutex-name ... 565
mutex-owner ... 402
mutex-specific ... 565
mutex-specific-set! ... 565
mutex-state ... 565
mutex-unlock! ... 565
mutex? ... 401
mv-call ... 747

N

n-for-each-par-map ... 411
n-par-for-each ... 411
n-par-map ... 411
nan ... 112
nan? ... 111, 599
native-endianness ... 162
native-eol-style ... 322
native-transcoder ... 323
negate ... 244
negative? ... 117, 596
netent:addrtype ... 489
netent:aliases ... 489
netent:name ... 489
netent:net ... 489
new-frame ... 746
newline ... 308, 613
next ... 50
next-token ... 672
next-token-of ... 672
nftw ... 647
ngettext ... 428
nice ... 478
ninth ... 536
nl_langinfo ... 424
no-applicable-method ... 699

no-infinities-violation? ... 618
no-method ... 699
no-nans-violation? ... 618
no-next-method ... 699
node-closure ... 676
node-eq? ... 675
node-equal? ... 675
node-join ... 676
node-or ... 676
node-parent ... 676
node-pos ... 675
node-reduce ... 676
node-reverse ... 675
node-self ... 676
node-trace ... 675
node-typeof? ... 675
nodeset? ... 675
non-continuable-violation? ... 610
nop ... 754
not ... 105, 593, 755
not-eq? ... 755
not-not ... 755
not-null? ... 755
not-pair? ... 536
ntohl ... 499
ntohs ... 499
null-environment ... 382, 625
null-list? ... 536
null-pointer? ... 389
null? ... 185, 593, 755
number->locale-string ... 424
number->string ... 117, 597
number? ... 106, 594
numerator ... 112, 596

O

objcode->bytecode ... 770
objcode? ... 769
object->string ... 270
object-properties ... 268
object-property ... 268
object-ref ... 752
object-stexi-documentation ... 683
odd? ... 115, 596
open ... 455
open-bytevector-input-port ... 326
open-bytevector-output-port ... 331
open-fdes ... 456
open-file ... 314
open-file-input-port ... 326
open-file-output-port ... 330
open-input-file ... 315, 612
open-input-output-pipe ... 483
open-input-pipe ... 483
open-input-string ... 318
open-output-file ... 315, 612
open-output-pipe ... 483
open-output-string ... 318
open-pipe ... 482
open-pipe* ... 482
open-server ... 523
open-socket-for-uri ... 522
opendir ... 464
optimize ... 48
option ... 50, 581
option-names ... 582
option-optional-arg? ... 582
option-processor ... 582
option-ref ... 532
option-required-arg? ... 582
or ... 283, 595
output-port? ... 305, 330, 612

P

package-stexi-documentation ... 684
package-stexi-documentation-for-include ... 684
package-stexi-extended-menu ... 684
package-stexi-generic-menu ... 683
package-stexi-standard-copying ... 683
package-stexi-standard-menu ... 684
package-stexi-standard-prologue ... 684
package-stexi-standard-titlepage ... 683
pair-fold ... 539
pair-fold-right ... 539
pair-for-each ... 541
pair? ... 182, 593, 755
par-for-each ... 411
par-map ... 411
parallel ... 411
parameterize ... 409
parse-c-struct ... 393
parse-header ... 508
parse-http-method ... 508
parse-http-version ... 508
parse-path ... 355
parse-request-uri ... 508
partial-cont-call ... 750
partition ... 542, 601
partition! ... 542
passwd:dir ... 466
passwd:gecos ... 466
passwd:gid ... 466
passwd:name ... 466
passwd:passwd ... 466
passwd:shell ... 466
passwd:uid ... 466
pause ... 481
pclose ... 483
peek-char ... 307, 613
peek-next-char ... 672
pipe ... 456
PKG_CHECK_MODULES ... 95
pointer->bytevector ... 390

pointer->procedure 393
pointer->scm 389
pointer->string 391
pointer-address 389
pointer? 389
popen 482
port->fdes 454
port->stream 651
port-closed? 310
port-column 308
port-conversion-strategy 306
port-encoding 305
port-eof? 325
port-filename 316
port-for-each 458
port-has-port-position? 325
port-has-set-port-position!? 325
port-line 308
port-mode 316, 458
port-position 325
port-revealed 454
port-transcoder 324
port-with-print-state 308
port? 305, 324
positive? 117, 596
post-order 674
pre-post-order 674
pretty-print 50, 632
primitive-_exit 477
primitive-eval 350
primitive-exit 477
primitive-fork 477
primitive-generic-generic 694
primitive-load 352
primitive-load-path 354
primitive-move->fdes 455
print-options 347
print-set! 348
procedure 49, 247
procedure->pointer 395
procedure-documentation 246
procedure-execution-count 449
procedure-name 245
procedure-properties 245
procedure-property 245
procedure-source 245
procedure-with-setter? 247
procedure? 245, 594
profile 49
program-arguments 471
program-arities 239
program-arity 239
program-bindings 239
program-free-variables 238
program-meta 239
program-module 238
program-objcode 238
program-objects 238
program-sources 239
program? 238
promise? 356
prompt 754
proper-list? 535
protoent:aliases 490
protoent:name 490
protoent:proto 490
provide 414
provided? 414
PTR2SCM 736
push-rest 748
put-bytevector 331
put-char 332
put-datum 332
put-string 332
put-u8 331
putenv 473
pwd 473

Q

q-empty-check 650
q-empty? 650
q-front 650
q-length 649
q-pop! 649
q-push! 649
q-rear 650
q-remove! 650
q? 649
quasiquote 344, 595
quasisyntax 620
quit 51
quo 756
quote 344, 595
quotient 116, 625

R

raise 479, 567, 608
raise-continuable 608
random 127
random-integer 576
random-real 576
random-source-make-integers 577
random-source-make-reals 578
random-source-pseudo-randomize! 577
random-source-randomize! 577
random-source-state-ref 577
random-source-state-set! 577
random-source? 577
random-state->datum 128
random-state-from-platform 128
random:exp 127
random:hollow-sphere! 127
random:normal 127
random:normal-vector! 127

random:solid-sphere! 128
random:uniform 128
rational-valued? 598
rational? 111, 596
rationalize 111, 596
re-export 370
read 346, 613
read-char 307, 613
read-client 523
read-delimited 311
read-delimited! 311
read-disable 347
read-enable 347
read-hash-extend 346
read-header 508
read-headers 508
read-line 311
read-line! 311
read-options 346
read-request 518
read-request-body 518
read-request-line 508
read-response 520
read-response-body 521
read-response-line 508
read-set! 347
read-string 672
read-string!/partial 312
read-text-line 672
read-with-shared-structure 583
readdir 464
readline 630
readline-disable 629
readline-enable 629
readline-options 629
readline-port 630
readline-set! 629
readlink 462
real->flonum 616
real-part 118, 595
real-valued? 598
real? 111, 596
rec 578
receive 291
record-accessor 213, 606
record-constructor 213, 606
record-constructor-descriptor 605
record-field-mutable? 607
record-modifier 213
record-mutator 606
record-predicate 213, 606
record-rtd 606
record-type-descriptor 213, 605
record-type-descriptor? 606
record-type-field-names 607
record-type-fields 214
record-type-generative? 607
record-type-name 214, 606
record-type-opaque? 607
record-type-parent 606
record-type-sealed? 607
record-type-uid 607
record? 212, 606
recv! 497
recvfrom! 498
redirect-port 457
reduce 539
reduce-right 539
regexp-exec 337
regexp-match? 340
regexp-quote 342
regexp-substitute 339
regexp-substitute/global 339
regexp? 338
registers 49
release-arbiter 396
release-port-handle 455
reload 48
reload-module 376
rem 756
remainder 116, 625
remove 542, 601
remove! 542
remove-class-accessors! 722
remove-hook! 272
remp 601
remq 601
remv 601
rename 463
rename-file 463
repl-set-default-option! 51
replace-range 674
replace-titles 678
request-absolute-uri 519
request-accept 519
request-accept-charset 519
request-accept-encoding 519
request-accept-language 519
request-allow 519
request-authorization 519
request-cache-control 519
request-connection 519
request-content-encoding 519
request-content-language 519
request-content-length 519
request-content-location 519
request-content-md5 519
request-content-range 519
request-content-type 519
request-date 519
request-expect 519
request-expires 519
request-from 519
request-headers 518
request-host 519
request-if-match 519

request-if-modified-since 519
request-if-none-match 519
request-if-range 519
request-if-unmodified-since 519
request-last-modified 519
request-max-forwards 519
request-meta 518
request-method 517
request-port 518
request-pragma 519
request-proxy-authorization 519
request-range 519
request-referer 519
request-te 519
request-trailer 519
request-transfer-encoding 519
request-upgrade 519
request-uri 517
request-user-agent 519
request-version 518
request-via 519
request-warning 519
request? 517
require 451
require-extension 585
reserve-locals 749
reset 288
reset-hook! 272
reset-parsed-entity-definitions! 671
resolve-interface 375
resolve-module 375
response-accept-ranges 521
response-age 521
response-allow 521
response-cache-control 521
response-code 520
response-connection 521
response-content-encoding 521
response-content-language 521
response-content-length 521
response-content-location 521
response-content-md5 521
response-content-range 521
response-content-type 521
response-date 521
response-etag 521
response-expires 521
response-headers 520
response-last-modified 521
response-location 521
response-must-not-include-body? 520
response-port 520
response-pragma 521
response-proxy-authenticate 521
response-reason-phrase 520
response-retry-after 521
response-server 521
response-trailer 521
response-transfer-encoding 521
response-upgrade 521
response-vary 521
response-version 520
response-via 521
response-warning 521
response-www-authenticate 521
response? 520
restore-signals 480
restricted-vector-sort! 269
return 747
return/nvalues 747
return/values 747
return/values* 747
reverse 188, 597
reverse! 188
reverse-bit-field 587
reverse-list->string 143
rewinddir 464
right-justify-string 681
rmdir 464
rotate-bit-field 587
round 120, 598
round-quotient 122
round-remainder 122
round/ 122
run-asyncs 398
run-hook 272
run-server 524

S

s16vector 551
s16vector->list 553
s16vector-length 552
s16vector-ref 552
s16vector-set! 553
s16vector? 550
s32vector 551
s32vector->list 553
s32vector-length 552
s32vector-ref 552
s32vector-set! 553
s32vector? 550
s64vector 551
s64vector->list 553
s64vector-length 552
s64vector-ref 552
s64vector-set! 553
s64vector? 550
s8vector 551
s8vector->list 553
s8vector-length 552
s8vector-ref 552
s8vector-set! 553
s8vector? 550
sanitize-response 524
save-module-excursion 375

scandir ... 647
scheme-report-environment ... 382, 625
scm->pointer ... 389
scm-error ... 297
scm_abs ... 119
scm_accept ... 497
scm_access ... 460
scm_acons ... 222
scm_add_feature ... 414
scm_add_hook_x ... 272
scm_alarm ... 480
scm_alignof ... 392
scm_all_threads ... 399
scm_angle ... 118
scm_any_to_c32vector ... 558
scm_any_to_c64vector ... 558
scm_any_to_f32vector ... 558
scm_any_to_f64vector ... 558
scm_any_to_s16vector ... 558
scm_any_to_s32vector ... 558
scm_any_to_s64vector ... 558
scm_any_to_s8vector ... 558
scm_any_to_u16vector ... 558
scm_any_to_u32vector ... 558
scm_any_to_u64vector ... 558
scm_any_to_u8vector ... 558
scm_append ... 187
scm_append_x ... 187
scm_apply ... 349
scm_apply_0 ... 349
scm_apply_1 ... 349
scm_apply_2 ... 349
scm_apply_3 ... 349
SCM_ARG1 ... 303
SCM_ARG2 ... 303
SCM_ARG3 ... 303
SCM_ARG4 ... 303
SCM_ARG5 ... 303
SCM_ARG6 ... 303
SCM_ARG7 ... 303
SCM_ARGn ... 303
scm_array_contents ... 205
scm_array_copy_x ... 202
scm_array_dimensions ... 201
scm_array_fill_x ... 202
scm_array_for_each ... 202
scm_array_get_handle ... 206
scm_array_handle_bit_elements ... 209
scm_array_handle_bit_writable_elements ... 210
scm_array_handle_c32_elements ... 209
scm_array_handle_c32_writable_elements ... 209
scm_array_handle_c64_elements ... 209
scm_array_handle_c64_writable_elements ... 209
scm_array_handle_dims ... 207
scm_array_handle_elements ... 208
scm_array_handle_f32_elements ... 208
scm_array_handle_f32_writable_elements ... 209
scm_array_handle_f64_elements ... 208
scm_array_handle_f64_writable_elements ... 209
scm_array_handle_pos ... 207
scm_array_handle_rank ... 206
scm_array_handle_ref ... 207
scm_array_handle_release ... 206
scm_array_handle_s16_elements ... 208
scm_array_handle_s16_writable_elements ... 209
scm_array_handle_s32_elements ... 208
scm_array_handle_s32_writable_elements ... 209
scm_array_handle_s64_elements ... 208
scm_array_handle_s64_writable_elements ... 209
scm_array_handle_s8_elements ... 208
scm_array_handle_s8_writable_elements ... 209
scm_array_handle_set ... 208
scm_array_handle_u16_elements ... 208
scm_array_handle_u16_writable_elements ... 209
scm_array_handle_u32_elements ... 208
scm_array_handle_u32_writable_elements ... 209
scm_array_handle_u64_elements ... 208
scm_array_handle_u64_writable_elements ... 209
scm_array_handle_u8_elements ... 208
scm_array_handle_u8_writable_elements ... 209
scm_array_handle_uniform_element_size ... 208
scm_array_handle_uniform_elements ... 208
scm_array_handle_uniform_writable_elements ... 208
scm_array_handle_writable_elements ... 208
scm_array_in_bounds_p ... 201
scm_array_index_map_x ... 202
scm_array_map_x ... 202
scm_array_p ... 200
scm_array_rank ... 201
scm_array_set_x ... 201
scm_array_to_list ... 202
scm_ash ... 125
SCM_ASSERT ... 303
scm_assert_smob_type ... 235
SCM_ASSERT_TYPE ... 303
scm_assoc ... 222
scm_assoc_ref ... 222
scm_assoc_remove_x ... 224
scm_assoc_set_x ... 222
scm_assq ... 222
scm_assq_ref ... 222
scm_assq_remove_x ... 224
scm_assq_set_x ... 222
scm_assv ... 222
scm_assv_ref ... 222
scm_assv_remove_x ... 224
scm_assv_set_x ... 222
scm_async ... 398
scm_async_mark ... 398
scm_backtrace ... 437
scm_backtrace_with_highlights ... 437
scm_basename ... 466
scm_bind ... 496
scm_bind_textdomain_codeset ... 429
scm_bindtextdomain ... 428

scm_bit_count ... 196
scm_bit_count_star ... 196
scm_bit_extract ... 127
scm_bit_invert_x ... 196
scm_bit_position ... 196
scm_bit_set_star_x ... 196
scm_bitvector ... 195
scm_bitvector_elements ... 197
scm_bitvector_fill_x ... 195
scm_bitvector_length ... 195
scm_bitvector_p ... 195
scm_bitvector_ref ... 195
scm_bitvector_set_x ... 195
scm_bitvector_to_list ... 195
scm_bitvector_writable_elements ... 197
scm_boolean_p ... 105
scm_boot_guile ... 102
scm_broadcast_condition_variable ... 403
scm_bytecode_to_objcode ... 769
SCM_BYTEVECTOR_CONTENTS ... 164
scm_bytevector_copy ... 163
scm_bytevector_copy_x ... 163
scm_bytevector_eq_p ... 163
scm_bytevector_fill_x ... 163
scm_bytevector_ieee_double_native_ref ... 167
scm_bytevector_ieee_double_native_set_x ... 167
scm_bytevector_ieee_double_ref ... 167
scm_bytevector_ieee_double_set_x ... 167
scm_bytevector_ieee_single_native_ref ... 167
scm_bytevector_ieee_single_native_set_x ... 167
scm_bytevector_ieee_single_ref ... 167
scm_bytevector_ieee_single_set_x ... 167
scm_bytevector_length ... 163
SCM_BYTEVECTOR_LENGTH ... 163
scm_bytevector_p ... 163
scm_bytevector_s16_native_ref ... 165
scm_bytevector_s16_native_set_x ... 166
scm_bytevector_s16_ref ... 165
scm_bytevector_s16_set_x ... 165
scm_bytevector_s32_native_ref ... 165
scm_bytevector_s32_native_set_x ... 166
scm_bytevector_s32_ref ... 165
scm_bytevector_s32_set_x ... 165
scm_bytevector_s64_native_ref ... 166
scm_bytevector_s64_native_set_x ... 166
scm_bytevector_s64_ref ... 165
scm_bytevector_s64_set_x ... 165
scm_bytevector_s8_ref ... 165
scm_bytevector_s8_set_x ... 165
scm_bytevector_sint_ref ... 164
scm_bytevector_sint_set_x ... 164
scm_bytevector_to_pointer ... 390
scm_bytevector_to_sint_list ... 166
scm_bytevector_to_u8_list ... 166
scm_bytevector_to_uint_list ... 166
scm_bytevector_u16_native_ref ... 165
scm_bytevector_u16_native_set_x ... 166
scm_bytevector_u16_ref ... 165
scm_bytevector_u16_set_x ... 165
scm_bytevector_u32_native_ref ... 165
scm_bytevector_u32_native_set_x ... 166
scm_bytevector_u32_ref ... 165
scm_bytevector_u32_set_x ... 165
scm_bytevector_u64_native_ref ... 165
scm_bytevector_u64_native_set_x ... 166
scm_bytevector_u64_ref ... 165
scm_bytevector_u64_set_x ... 165
scm_bytevector_u8_ref ... 165
scm_bytevector_u8_set_x ... 165
scm_bytevector_uint_ref ... 164
scm_bytevector_uint_set_x ... 164
scm_c_angle ... 119
scm_c_array_rank ... 202
scm_c_bitvector_length ... 195
scm_c_bitvector_ref ... 195
scm_c_bitvector_set_x ... 195
scm_c_bytevector_length ... 163
scm_c_bytevector_ref ... 163
scm_c_bytevector_set_x ... 163
scm_c_call_with_blocked_asyncs ... 398
scm_c_call_with_current_module ... 376
scm_c_call_with_unblocked_asyncs ... 398
scm_c_catch ... 294
scm_c_define ... 276, 378
scm_c_define_gsubr ... 237
scm_c_define_module ... 378
scm_c_downcase ... 133
scm_c_eval_string ... 349
scm_c_export ... 379
scm_c_generalized_vector_length ... 197
scm_c_generalized_vector_ref ... 197
scm_c_generalized_vector_set_x ... 198
scm_c_hook_add ... 274
scm_c_hook_init ... 274
scm_c_hook_remove ... 274
scm_c_hook_run ... 274
scm_c_imag_part ... 118
scm_c_locale_stringn_to_number ... 118
scm_c_lookup ... 377
scm_c_magnitude ... 119
scm_c_make_bitvector ... 195
scm_c_make_bytevector ... 163
scm_c_make_gsubr ... 237
scm_c_make_polar ... 118
scm_c_make_rectangular ... 118
scm_c_make_socket_address ... 493
scm_c_make_string ... 143
scm_c_make_vector ... 191
scm_c_module_define ... 378
scm_c_module_lookup ... 378
scm_c_nvalues ... 290
scm_c_port_for_each ... 458
scm_c_primitive_load ... 353
scm_c_private_lookup ... 377

scm_c_private_ref 377
scm_c_private_variable 377
scm_c_public_lookup 377
scm_c_public_ref 377
scm_c_public_variable 376
scm_c_read 307
scm_c_real_part 118
scm_c_resolve_module 379
scm_c_round 120
scm_c_run_hook 273
scm_c_string_length 144
scm_c_string_ref 145
scm_c_string_set_x 147
scm_c_substring 145
scm_c_substring_copy 145
scm_c_substring_read_only 145
scm_c_substring_shared 145
scm_c_symbol_length 174
scm_c_titlecase 133
scm_c_truncate 120
scm_c_uniform_vector_length 556
scm_c_upcase 133
scm_c_use_module 379
scm_c_value_ref 290
scm_c_values 290
scm_c_vector_length 192
scm_c_vector_ref 192
scm_c_vector_set_x 192
scm_c_with_continuation_barrier 304
scm_c_with_dynamic_state 408
scm_c_with_fluid 407
scm_c_with_fluids 407
scm_c_with_throw_handler 295
scm_c_write 309
scm_c32vector 552
scm_c32vector_elements 555
scm_c32vector_length 552
scm_c32vector_p 550
scm_c32vector_ref 553
scm_c32vector_set_x 553
scm_c32vector_to_list 554
scm_c32vector_writable_elements 556
scm_c64vector 552
scm_c64vector_elements 555
scm_c64vector_length 552
scm_c64vector_p 550
scm_c64vector_ref 553
scm_c64vector_set_x 553
scm_c64vector_to_list 554
scm_c64vector_writable_elements 556
scm_caaaar 184
scm_caaadr 184
scm_caaar 184
scm_caadar 184
scm_caaddr 184
scm_caadr 184
scm_caar 184
scm_cadaar 184
scm_cadadr 184
scm_cadar 184
scm_caddar 184
scm_cadddr 184
scm_caddr 184
scm_cadr 184
scm_call 349
scm_call_0 349
scm_call_1 349
scm_call_2 349
scm_call_3 349
scm_call_4 349
scm_call_5 349
scm_call_6 349
scm_call_7 349
scm_call_8 349
scm_call_9 349
scm_call_n 349
scm_call_with_blocked_asyncs 397
scm_call_with_input_string 317
scm_call_with_output_string 317
scm_call_with_unblocked_asyncs 398
scm_calloc 360
scm_cancel_thread 400
scm_car 183
SCM_CAR 183
scm_catch 293
scm_catch_with_pre_unwind_handler 293
scm_cdaaar 184
scm_cdaadr 184
scm_cdaar 184
scm_cdadar 184
scm_cdaddr 184
scm_cdadr 184
scm_cdar 184
scm_cddaar 184
scm_cddadr 184
scm_cddar 184
scm_cdddar 184
scm_cddddr 184
scm_cdddr 184
scm_cddr 184
scm_cdr 183
SCM_CDR 183
scm_ceiling 120
scm_ceiling_divide 121
scm_ceiling_quotient 121
scm_ceiling_remainder 121
scm_cell 736
SCM_CELL_OBJECT 737
SCM_CELL_TYPE 737
SCM_CELL_WORD 737
scm_centered_divide 122
scm_centered_quotient 122
scm_centered_remainder 122
scm_char_alphabetic_p 131
scm_char_downcase 132
scm_char_general_category 132

scm_char_is_both_p 131
scm_char_locale_ci_eq 422
scm_char_locale_ci_gt 422
scm_char_locale_ci_lt 422
scm_char_locale_downcase 423
scm_char_locale_gt 422
scm_char_locale_lt 422
scm_char_locale_titlecase 423
scm_char_locale_upcase 423
scm_char_lower_case_p 131
scm_char_numeric_p 131
scm_char_p 130
scm_char_ready_p 306
scm_char_set 135
scm_char_set_adjoin 137
scm_char_set_adjoin_x 137
scm_char_set_any 137
scm_char_set_complement 137
scm_char_set_complement_x 138
scm_char_set_contains_p 137
scm_char_set_copy 135
scm_char_set_count 136
scm_char_set_cursor 133
scm_char_set_cursor_next 134
scm_char_set_delete 137
scm_char_set_delete_x 137
scm_char_set_diff_plus_intersection 138
scm_char_set_diff_plus_intersection_x ... 138
scm_char_set_difference 138
scm_char_set_difference_x 138
scm_char_set_eq 133
scm_char_set_every 137
scm_char_set_filter 135
scm_char_set_filter_x 135
scm_char_set_fold 134
scm_char_set_for_each 134
scm_char_set_hash 133
scm_char_set_intersection 138
scm_char_set_intersection_x 138
scm_char_set_leq 133
scm_char_set_map 134
scm_char_set_p 133
scm_char_set_ref 134
scm_char_set_size 136
scm_char_set_to_list 136
scm_char_set_to_string 136
scm_char_set_unfold 134
scm_char_set_unfold_x 134
scm_char_set_union 137
scm_char_set_union_x 138
scm_char_set_xor 138
scm_char_set_xor_x 138
scm_char_titlecase 132
scm_char_to_integer 132
scm_char_upcase 132
scm_char_upper_case_p 131
scm_char_whitespace_p 131
scm_chdir 473
scm_chmod 463
scm_chown 463
scm_chroot 473
scm_close 456
scm_close_fdes 456
scm_close_input_port 309
scm_close_output_port 309
scm_close_port 309
scm_closedir 464
scm_complex_p 113
scm_condition_variable_p 403
scm_connect 496
scm_cons 182
scm_cons_source 433
scm_copy_file 463
scm_copy_random_state 127
scm_copy_tree 270
SCM_CRITICAL_SECTION_END 405
SCM_CRITICAL_SECTION_START 405
scm_crypt 501
scm_ctermid 482
scm_current_dynamic_state 407
scm_current_error_port 314
scm_current_input_port 313
scm_current_load_port 353
scm_current_module 375
scm_current_output_port 313
scm_current_processor_count 479
scm_current_thread 399
scm_current_time 468
scm_datum_to_random_state 128
scm_define 276, 378
SCM_DEFINE 102
scm_defined_p 279
scm_delete 188
scm_delete_file 463
scm_delete_x 188
scm_delete1_x 189
scm_delq 188
scm_delq_x 188
scm_delq1_x 189
scm_delv 188
scm_delv_x 188
scm_delv1_x 189
scm_denominator 112
scm_difference 119
scm_directory_stream_p 464
scm_dirname 466
scm_display_application 432
scm_display_backtrace 431
scm_display_backtrace_with_highlights ... 431
scm_display_error 302
scm_divide 119
scm_done_free 362
scm_done_malloc 362
scm_double_cell 736
scm_doubly_weak_hash_table_p 363
scm_drain_input 308

scm_dup_to_fdes 457
scm_dup2 458
scm_dynamic_call 384
scm_dynamic_func 384
scm_dynamic_link 383
scm_dynamic_object_p 383
scm_dynamic_pointer 389
scm_dynamic_state_p 407
scm_dynamic_unlink 383
scm_dynamic_wind 299
scm_dynwind_begin 300
scm_dynwind_block_asyncs 398
scm_dynwind_critical_section 405
scm_dynwind_current_dynamic_state 408
scm_dynwind_current_error_port 314
scm_dynwind_current_input_port 314
scm_dynwind_current_output_port 314
scm_dynwind_end 300
scm_dynwind_fluid 407
scm_dynwind_free 301, 361
scm_dynwind_lock_mutex 402
scm_dynwind_rewind_handler 301
scm_dynwind_rewind_handler_with_scm 301
scm_dynwind_unblock_asyncs 398
scm_dynwind_unwind_handler 301
scm_dynwind_unwind_handler_with_scm 301
scm_effective_version 412
scm_end_of_char_set_p 134
scm_environ 473
scm_eof_object 324
scm_eof_object_p 306, 324
SCM_EOF_VAL 735
SCM_EOL 735
scm_eq_p 265
scm_equal_p 266
scm_eqv_p 266
scm_error 302
scm_error_scm 297
scm_euclidean_divide 120
scm_euclidean_quotient 120
scm_euclidean_remainder 120
scm_eval 348
scm_eval_string 349
scm_eval_string_in_module 349
scm_even_p 115
scm_exact_integer_sqrt 116
scm_exact_p 113
scm_exact_to_inexact 114
scm_execl 477
scm_execle 477
scm_execlp 477
scm_f32vector 552
scm_f32vector_elements 555
scm_f32vector_length 552
scm_f32vector_p 550
scm_f32vector_ref 553
scm_f32vector_set_x 553
scm_f32vector_to_list 554
scm_f32vector_writable_elements 556
scm_f64vector 552
scm_f64vector_elements 555
scm_f64vector_length 552
scm_f64vector_p 550
scm_f64vector_ref 553
scm_f64vector_set_x 553
scm_f64vector_to_list 554
scm_f64vector_writable_elements 556
scm_fcntl 458
scm_fdes_to_ports 455
scm_fdopen 454
scm_file_encoding 356
scm_file_port_p 316
scm_fileno 454
scm_finite_p 112
scm_flock 459
scm_floor 120
scm_floor_divide 120
scm_floor_quotient 120
scm_floor_remainder 121
scm_fluid_bound_p 406
scm_fluid_p 406
scm_fluid_ref 406
scm_fluid_set_x 406
scm_fluid_unset_x 406
scm_flush_all_ports 309
scm_force 356
scm_force_output 309
scm_fork 477
scm_frame_arguments 431
scm_frame_p 431
scm_frame_previous 431
scm_frame_procedure 431
scm_from_bool 105
scm_from_char 109
scm_from_double 112
scm_from_int 109
scm_from_int16 109
scm_from_int32 109
scm_from_int64 109
scm_from_int8 109
scm_from_intmax 110
scm_from_latin1_keyword 181
scm_from_latin1_string 160
scm_from_latin1_stringn 160
scm_from_locale_keyword 181
scm_from_locale_keywordn 181
scm_from_locale_string 158
scm_from_locale_stringn 158
scm_from_long 109
scm_from_long_long 109
scm_from_mpz 110
scm_from_pointer 390
scm_from_schar 109
scm_from_short 109
scm_from_signed_integer 108
scm_from_size_t 109

scm_from_sockaddr 493
scm_from_ssize_t 109
scm_from_stringn 160
scm_from_uchar 109
scm_from_uint 109
scm_from_uint16 109
scm_from_uint32 109
scm_from_uint64 109
scm_from_uint8 109
scm_from_uintmax 110
scm_from_ulong 109
scm_from_ulong_long 109
scm_from_unsigned_integer 108
scm_from_ushort 109
scm_from_utf32_string 160
scm_from_utf32_stringn 160
scm_from_utf8_keyword 181
scm_from_utf8_string 160
scm_from_utf8_stringn 160
scm_fsync 455
scm_ftell 310
scm_gc 358
scm_gc_calloc 361
scm_gc_free 361
scm_gc_live_object_stats 359
scm_gc_malloc 360
scm_gc_malloc_pointerless 360
scm_gc_mark 359
scm_gc_protect_object 358
scm_gc_realloc 360
scm_gc_register_allocation 361
scm_gc_stats 359
scm_gc_unprotect_object 359
scm_gcd 116
scm_generalized_vector_get_handle 198
scm_generalized_vector_length 197
scm_generalized_vector_p 197
scm_generalized_vector_ref 197
scm_generalized_vector_set_x 197
scm_generalized_vector_to_list 197
scm_gensym 174
scm_geq_p 117
scm_get_bytevector_all 328
scm_get_bytevector_n 328
scm_get_bytevector_n_x 328
scm_get_bytevector_some 328
scm_get_internal_real_time 471
scm_get_internal_run_time 471
scm_get_output_string 318
scm_get_print_state 308
scm_get_u8 328
scm_getaddrinfo 485
scm_getaffinity 478
scm_getcwd 473
scm_getegid 474
scm_getenv 472
scm_geteuid 474
scm_getgid 474
scm_getgrgid 468
scm_getgroups 473
scm_gethost 488
scm_gethostname 501
scm_getitimer 481
scm_getlogin 468
scm_getnet 489
scm_getpass 502
scm_getpeername 497
scm_getpgrp 475
scm_getpid 473
scm_getppid 474
scm_getpriority 478
scm_getproto 490
scm_getpwuid 467
scm_getserv 491
scm_getsid 475
scm_getsockname 497
scm_getsockopt 494
scm_gettext 427
scm_gettimeofday 468
scm_getuid 474
SCM_GLOBAL_KEYWORD 103
SCM_GLOBAL_SYMBOL 103
SCM_GLOBAL_VARIABLE 103
SCM_GLOBAL_VARIABLE_INIT 103
scm_gmtime 469
scm_gr_p 117
scm_hash 232
scm_hash_clear_x 231
scm_hash_create_handle_x 232
scm_hash_fold 233
scm_hash_for_each 232
scm_hash_for_each_handle 233
scm_hash_get_handle 232
scm_hash_map_to_list 232
scm_hash_ref 231
scm_hash_remove_x 231
scm_hash_set_x 231
scm_hash_table_p 231
scm_hashq 232
scm_hashq_create_handle_x 232
scm_hashq_get_handle 232
scm_hashq_ref 231
scm_hashq_remove_x 231
scm_hashq_set_x 231
scm_hashv 232
scm_hashv_create_handle_x 232
scm_hashv_get_handle 232
scm_hashv_ref 231
scm_hashv_remove_x 231
scm_hashv_set_x 231
scm_hashx_create_handle_x 232
scm_hashx_get_handle 232
scm_hashx_ref 231
scm_hashx_remove_x 231
scm_hashx_set_x 231
scm_hook_empty_p 272

scm_hook_p ... 272
scm_hook_to_list ... 272
SCM_HOOKP ... 273
scm_htonl ... 499
scm_htons ... 499
scm_imag_part ... 118
SCM_IMP ... 735
scm_inet_aton ... 484
scm_inet_makeaddr ... 485
scm_inet_netof ... 484
scm_inet_ntoa ... 484
scm_inet_ntop ... 485
scm_inet_pton ... 485
scm_inexact_p ... 114
scm_inexact_to_exact ... 114
scm_inf ... 112
scm_inf_p ... 111
scm_init_guile ... 101
scm_input_port_p ... 305
scm_integer_expt ... 126
scm_integer_length ... 126
scm_integer_p ... 107
scm_integer_to_char ... 132
scm_interaction_environment ... 348
scm_internal_catch ... 294
scm_is_array ... 200
scm_is_bitvector ... 195
scm_is_bool ... 105
scm_is_bytevector ... 163
scm_is_complex ... 113
scm_is_dynamic_state ... 407
scm_is_eq ... 266
scm_is_exact ... 113
scm_is_false ... 105
scm_is_generalized_vector ... 197
scm_is_inexact ... 114
scm_is_integer ... 108
scm_is_keyword ... 181
scm_is_null ... 186
scm_is_number ... 106
scm_is_pair ... 183
scm_is_rational ... 112
scm_is_real ... 112
scm_is_signed_integer ... 108
scm_is_simple_vector ... 193
scm_is_string ... 142
scm_is_symbol ... 172
scm_is_true ... 105
scm_is_typed_array ... 200
scm_is_uniform_vector ... 556
scm_is_unsigned_integer ... 108
scm_is_vector ... 191
scm_isatty_p ... 482
scm_join_thread ... 399
scm_join_thread_timed ... 399
SCM_KEYWORD ... 103
scm_keyword_p ... 181
scm_keyword_to_symbol ... 181
scm_kill ... 479
scm_last_pair ... 187
scm_lcm ... 116
scm_length ... 186
scm_leq_p ... 117
scm_less_p ... 117
scm_link ... 464
scm_list_1 ... 186
scm_list_2 ... 186
scm_list_3 ... 186
scm_list_4 ... 186
scm_list_5 ... 186
scm_list_cdr_set_x ... 188
scm_list_copy ... 186
scm_list_head ... 187
scm_list_n ... 186
scm_list_p ... 185
scm_list_ref ... 187
scm_list_set_x ... 188
scm_list_tail ... 187
scm_list_to_bitvector ... 195
scm_list_to_c32vector ... 554
scm_list_to_c64vector ... 554
scm_list_to_char_set ... 135
scm_list_to_char_set_x ... 135
scm_list_to_f32vector ... 554
scm_list_to_f64vector ... 554
scm_list_to_s16vector ... 554
scm_list_to_s32vector ... 554
scm_list_to_s64vector ... 554
scm_list_to_s8vector ... 554
scm_list_to_typed_array ... 201
scm_list_to_u16vector ... 554
scm_list_to_u32vector ... 554
scm_list_to_u64vector ... 554
scm_list_to_u8vector ... 554
scm_listen ... 496
scm_lnaof ... 485
scm_load_extension ... 384
scm_load_objcode ... 769
scm_local_eval ... 357
scm_locale_p ... 421
scm_locale_string_to_inexact ... 424
scm_locale_string_to_integer ... 423
scm_localtime ... 469
scm_lock_mutex ... 401
scm_lock_mutex_timed ... 401
scm_logand ... 124
scm_logbit_p ... 125
scm_logcount ... 126
scm_logior ... 125
scm_lognot ... 125
scm_logtest ... 125
scm_lookahead_u8 ... 328
scm_lookup ... 378
scm_loxor ... 125
scm_lstat ... 462
scm_macro_binding ... 265

scm_macro_name ... 265
scm_macro_p ... 264
scm_macro_transformer ... 265
scm_macro_type ... 264
scm_magnitude ... 118
scm_major_version ... 412
scm_make_arbiter ... 396
scm_make_array ... 200
scm_make_bitvector ... 195
scm_make_bytevector ... 163
scm_make_c32vector ... 551
scm_make_c64vector ... 551
scm_make_condition_variable ... 402
scm_make_custom_binary_input_port ... 327
scm_make_custom_binary_output_port ... 331
scm_make_doubly_weak_hash_table ... 363
scm_make_dynamic_state ... 407
scm_make_f32vector ... 551
scm_make_f64vector ... 551
scm_make_fluid ... 406
scm_make_fluid_with_default ... 406
scm_make_guardian ... 364
scm_make_hook ... 272
scm_make_locale ... 421
scm_make_mutex ... 401
scm_make_mutex_with_flags ... 401
scm_make_polar ... 118
scm_make_port_type ... 334
scm_make_procedure_with_setter ... 247
scm_make_program ... 770
scm_make_rectangular ... 118
scm_make_recursive_mutex ... 401
scm_make_regexp ... 337
scm_make_s16vector ... 551
scm_make_s32vector ... 551
scm_make_s64vector ... 551
scm_make_s8vector ... 551
scm_make_shared_array ... 203
scm_make_smob_type ... 233
scm_make_socket_address ... 492
scm_make_soft_port ... 318
scm_make_stack ... 430
scm_make_string ... 143
scm_make_struct ... 215
scm_make_struct_layout ... 218
scm_make_symbol ... 177
scm_make_typed_array ... 200
scm_make_u16vector ... 551
scm_make_u32vector ... 551
scm_make_u64vector ... 551
scm_make_u8vector ... 551
scm_make_unbound_fluid ... 406
scm_make_undefined_variable ... 374
scm_make_variable ... 374
scm_make_vector ... 191
scm_make_vtable_vtable ... 217
scm_make_weak_key_hash_table ... 363
scm_make_weak_value_hash_table ... 363
scm_make_weak_vector ... 363
scm_malloc ... 360
scm_map ... 190
scm_markcdr ... 236
scm_max ... 119
scm_member ... 189
scm_memory_error ... 302
scm_memq ... 189
scm_memv ... 189
scm_merge ... 268
scm_merge_x ... 268
scm_micro_version ... 412
scm_min ... 119
scm_minor_version ... 412
scm_misc_error ... 303
scm_mkdir ... 464
scm_mknod ... 465
scm_mkstemp ... 465
scm_mktime ... 470
scm_module_define ... 378
scm_module_ensure_local_variable ... 378
scm_module_lookup ... 378
scm_module_reverse_lookup ... 378
scm_module_variable ... 378
scm_modulo ... 116
scm_modulo_expt ... 116
scm_must_calloc ... 361
scm_must_free ... 361
scm_must_malloc ... 361
scm_must_realloc ... 361
scm_mutex_level ... 402
scm_mutex_locked_p ... 402
scm_mutex_owner ... 402
scm_mutex_p ... 401
scm_nan ... 112
scm_nan_p ... 111
scm_native_endianness ... 162
scm_nconc2last ... 349
scm_negative_p ... 117
scm_newline ... 308
SCM_NEWSMOB ... 235
SCM_NEWSMOB2 ... 235
SCM_NEWSMOB3 ... 235
scm_ngettext ... 428
scm_nice ... 478
scm_not ... 105
scm_ntohl ... 499
scm_ntohs ... 499
scm_null_p ... 185
scm_num_eq_p ... 117
scm_num_overflow ... 302
scm_number_p ... 106
scm_number_to_string ... 117
scm_numerator ... 112
scm_objcode_p ... 769
scm_objcode_to_bytecode ... 770
scm_object_properties ... 268
scm_object_property ... 268

scm_object_to_string 270
scm_odd_p 115
scm_oneminus 119
scm_oneplus 119
scm_open 455
scm_open_bytevector_input_port 326
scm_open_bytevector_output_port 331
scm_open_fdes 456
scm_open_file 314
scm_open_input_string 318
scm_open_output_string 318
scm_opendir 464
scm_out_of_range 302
scm_output_port_p 305
SCM_PACK 100
scm_pair_p 182
scm_parse_path 355
scm_pause 481
scm_peek_char 307
scm_permanent_object 359
scm_pipe 456
scm_pointer_address 389
scm_pointer_to_bytevector 390
scm_pointer_to_procedure 393
scm_port_closed_p 310
scm_port_column 308
scm_port_conversion_strategy 306
scm_port_encoding 305
scm_port_filename 316
scm_port_for_each 458
scm_port_line 308
scm_port_mode 316
scm_port_p 305
scm_port_revealed 454
scm_port_with_print_state 308
scm_positive_p 117
scm_primitive__exit 477
scm_primitive_eval 350
scm_primitive_exit 477
scm_primitive_load 352
scm_primitive_load_path 354
scm_primitive_move_to_fdes 455
scm_private_lookup 377
scm_private_ref 377
scm_private_variable 377
scm_procedure 247
scm_procedure_documentation 246
scm_procedure_name 245
scm_procedure_p 245
scm_procedure_properties 245
scm_procedure_property 245
scm_procedure_source 245
scm_procedure_to_pointer 395
scm_procedure_with_setter_p 247
scm_product 119
scm_program_arguments 471
scm_program_arities 239
scm_program_free_variables 238
scm_program_meta 239
scm_program_module 238
scm_program_objcode 238
scm_program_objects 238
scm_program_p 238
scm_promise_p 356
SCM_PTAB_ENTRY 332
scm_pthread_cond_timedwait 404
scm_pthread_cond_wait 404
scm_pthread_mutex_lock 404
SCM_PTOBNUM 332
scm_public_lookup 377
scm_public_ref 377
scm_public_variable 376
scm_put_bytevector 331
scm_put_u8 331
scm_putenv 473
scm_quotient 116
scm_raise 479
scm_random 127
scm_random_exp 127
scm_random_hollow_sphere_x 127
scm_random_normal 127
scm_random_normal_vector_x 127
scm_random_solid_sphere_x 128
scm_random_state_from_platform 128
scm_random_state_to_datum 128
scm_random_uniform 128
scm_rational_p 111
scm_rationalize 111
scm_read 346
scm_read_char 307
scm_read_delimited_x 312
scm_read_hash_extend 346
scm_read_line 312
scm_read_string_x_partial 312
scm_readdir 464
scm_readlink 462
scm_real_p 111
scm_real_part 118
scm_realloc 360
scm_recv 497
scm_recvfrom 498
scm_redirect_port 457
scm_regexp_exec 337
scm_regexp_p 338
scm_release_arbiter 396
scm_remainder 116
scm_remember_upto_here_1 359
scm_remember_upto_here_2 359
scm_remove_hook_x 272
scm_rename 463
scm_reset_hook_x 272
scm_resolve_module 375
scm_restore_signals 480
scm_restricted_vector_sort_x 269
SCM_RETURN_NEWSMOB 235
SCM_RETURN_NEWSMOB2 235

SCM_RETURN_NEWSMOB3 235
scm_reverse 188
scm_reverse_list_to_string 143
scm_reverse_x 188
scm_rewinddir 464
scm_rmdir 464
scm_round_divide 122
scm_round_number 120
scm_round_quotient 122
scm_round_remainder 122
scm_run_asyncs 398
scm_run_hook 272
scm_s16vector 551
scm_s16vector_elements 555
scm_s16vector_length 552
scm_s16vector_p 550
scm_s16vector_ref 553
scm_s16vector_set_x 553
scm_s16vector_to_list 554
scm_s16vector_writable_elements 556
scm_s32vector 551
scm_s32vector_elements 555
scm_s32vector_length 552
scm_s32vector_p 550
scm_s32vector_ref 553
scm_s32vector_set_x 553
scm_s32vector_to_list 554
scm_s32vector_writable_elements 556
scm_s64vector 552
scm_s64vector_elements 555
scm_s64vector_length 552
scm_s64vector_p 550
scm_s64vector_ref 553
scm_s64vector_set_x 553
scm_s64vector_to_list 554
scm_s64vector_writable_elements 556
scm_s8vector 551
scm_s8vector_elements 555
scm_s8vector_length 552
scm_s8vector_p 550
scm_s8vector_ref 553
scm_s8vector_set_x 553
scm_s8vector_to_list 554
scm_s8vector_writable_elements 556
scm_search_path 355
scm_seed_to_random_state 128
scm_seek 310
scm_select 460
scm_send 497
scm_sendto 498
scm_set_car_x 184
scm_set_cdr_x 184
SCM_SET_CELL_OBJECT 738
SCM_SET_CELL_TYPE 737
SCM_SET_CELL_WORD 737
scm_set_current_dynamic_state 407
scm_set_current_error_port 314
scm_set_current_input_port 314
scm_set_current_module 375
scm_set_current_output_port 314
scm_set_object_properties_x 268
scm_set_object_property_x 268
scm_set_port_close 334
scm_set_port_column_x 308
scm_set_port_conversion_strategy_x 306
scm_set_port_encoding_x 305
scm_set_port_end_input 335
scm_set_port_equalp 334
scm_set_port_filename_x 316
scm_set_port_flush 335
scm_set_port_free 334
scm_set_port_input_waiting 335
scm_set_port_line_x 308
scm_set_port_mark 334
scm_set_port_print 334
scm_set_port_revealed_x 454
scm_set_port_seek 335
scm_set_port_truncate 335
scm_set_procedure_properties_x 246
scm_set_procedure_property_x 246
scm_set_program_arguments 472
scm_set_program_arguments_scm 471
SCM_SET_SMOB_DATA 235
SCM_SET_SMOB_DATA_2 235
SCM_SET_SMOB_DATA_3 235
scm_set_smob_equalp 234
SCM_SET_SMOB_FLAGS 235
scm_set_smob_free 234
scm_set_smob_mark 234
SCM_SET_SMOB_OBJECT 236
SCM_SET_SMOB_OBJECT_2 236
SCM_SET_SMOB_OBJECT_3 236
scm_set_smob_print 234
scm_set_source_properties_x 433
scm_set_source_property_x 433
scm_set_struct_vtable_name_x 217
scm_set_thread_cleanup_x 400
scm_setaffinity 478
scm_setegid 475
scm_seteuid 474
scm_setgid 474
scm_setgrent 468
scm_setgroups 474
scm_sethost 489
scm_sethostname 501
scm_setitimer 481
scm_setlocale 501
scm_setnet 490
scm_setpgid 475
scm_setpriority 478
scm_setproto 491
scm_setpwent 467
scm_setserv 492
scm_setsid 475
scm_setsockopt 494
scm_setuid 474

scm_setvbuf ... 458
scm_shared_array_increments ... 205
scm_shared_array_offset ... 205
scm_shared_array_root ... 205
scm_shell ... 102
scm_shutdown ... 496
scm_sigaction ... 480
scm_sigaction_for_thread ... 480
scm_signal_condition_variable ... 403
scm_simple_format ... 309
SCM_SIMPLE_VECTOR_LENGTH ... 193
SCM_SIMPLE_VECTOR_REF ... 193
SCM_SIMPLE_VECTOR_SET ... 193
scm_sint_list_to_bytevector ... 167
scm_sizeof ... 392
scm_sleep ... 481
scm_sloppy_assoc ... 225
scm_sloppy_assq ... 224
scm_sloppy_assv ... 225
SCM_SMOB_DATA ... 235
SCM_SMOB_DATA_2 ... 235
SCM_SMOB_DATA_3 ... 235
SCM_SMOB_FLAGS ... 235
SCM_SMOB_OBJECT ... 236
SCM_SMOB_OBJECT_2 ... 236
SCM_SMOB_OBJECT_2_LOC ... 236
SCM_SMOB_OBJECT_3 ... 236
SCM_SMOB_OBJECT_3_LOC ... 236
SCM_SMOB_OBJECT_LOC ... 236
SCM_SMOB_PREDICATE ... 235
SCM_SNARF_INIT ... 102
scm_socket ... 494
scm_socketpair ... 494
scm_sort ... 269
scm_sort_list ... 269
scm_sort_list_x ... 269
scm_sort_x ... 269
scm_sorted_p ... 269
scm_source_properties ... 433
scm_source_property ... 433
scm_spawn_thread ... 399
scm_stable_sort ... 269
scm_stable_sort_x ... 269
scm_stack_id ... 431
scm_stack_length ... 431
scm_stack_p ... 431
scm_stack_ref ... 431
scm_stat ... 461
scm_status_exit_val ... 476
scm_status_stop_sig ... 476
scm_status_term_sig ... 476
scm_std_select ... 404
scm_std_sleep ... 404
scm_std_usleep ... 404
scm_strerror ... 297
scm_strftime ... 470
scm_string ... 143
scm_string_any ... 142
scm_string_append ... 154
scm_string_append_shared ... 155
scm_string_bytes_per_char ... 161
scm_string_capitalize ... 154
scm_string_capitalize_x ... 154
scm_string_ci_eq ... 149
scm_string_ci_ge ... 150
scm_string_ci_gt ... 149
scm_string_ci_le ... 150
scm_string_ci_lt ... 149
scm_string_ci_neq ... 149
scm_string_ci_to_symbol ... 173
scm_string_compare ... 148
scm_string_compare_ci ... 149
scm_string_concatenate ... 155
scm_string_concatenate_reverse ... 155
scm_string_concatenate_reverse_shared ... 155
scm_string_concatenate_shared ... 155
scm_string_contains ... 153
scm_string_contains_ci ... 153
scm_string_copy ... 145
scm_string_copy_x ... 147
scm_string_count ... 153
scm_string_delete ... 157
scm_string_downcase ... 153
scm_string_downcase_x ... 154
scm_string_drop ... 145
scm_string_drop_right ... 146
scm_string_eq ... 149
scm_string_every ... 142
scm_string_fill_x ... 147
scm_string_filter ... 157
scm_string_fold ... 156
scm_string_fold_right ... 156
scm_string_for_each ... 155
scm_string_for_each_index ... 155
scm_string_ge ... 149
scm_string_gt ... 149
scm_string_index ... 151
scm_string_index_right ... 152
scm_string_join ... 143
scm_string_le ... 149
scm_string_length ... 144
scm_string_locale_ci_eq ... 422
scm_string_locale_ci_gt ... 422
scm_string_locale_ci_lt ... 422
scm_string_locale_downcase ... 423
scm_string_locale_gt ... 422
scm_string_locale_lt ... 422
scm_string_locale_titlecase ... 423
scm_string_locale_upcase ... 423
scm_string_lt ... 149
scm_string_map ... 155
scm_string_map_x ... 155
scm_string_neq ... 149
scm_string_normalize_nfc ... 151
scm_string_normalize_nfd ... 151
scm_string_normalize_nfkc ... 151

scm_string_normalize_nfkd 151
scm_string_null_p 142
scm_string_p 142
scm_string_pad 146
scm_string_pad_right 146
scm_string_prefix_ci_p 152
scm_string_prefix_length 151
scm_string_prefix_length_ci 151
scm_string_prefix_p 152
scm_string_ref 144
scm_string_replace 157
scm_string_reverse 154
scm_string_reverse_x 154
scm_string_rindex 151
scm_string_set_x 146
scm_string_skip 152
scm_string_skip_right 152
scm_string_split 144
scm_string_suffix_ci_p 152
scm_string_suffix_length 151
scm_string_suffix_length_ci 152
scm_string_suffix_p 152
scm_string_tabulate 143
scm_string_take 145
scm_string_take_right 145
scm_string_titlecase 154
scm_string_titlecase_x 154
scm_string_to_char_set 135
scm_string_to_char_set_x 135
scm_string_to_list 144
scm_string_to_number 118
scm_string_to_symbol 173
scm_string_to_utf16 168
scm_string_to_utf32 168
scm_string_to_utf8 168
scm_string_tokenize 157
scm_string_trim 146
scm_string_trim_both 146
scm_string_trim_right 146
scm_string_unfold 156
scm_string_unfold_right 156
scm_string_upcase 153
scm_string_upcase_x 153
scm_string_xcopy_x 157
scm_strptime 470
scm_struct_p 216
scm_struct_ref 216
scm_struct_set_x 216
scm_struct_vtable 216
scm_struct_vtable_name 217
scm_struct_vtable_p 216
scm_substring 145
scm_substring_copy 145
scm_substring_downcase 153
scm_substring_downcase_x 154
scm_substring_fill_x 147
scm_substring_hash 150
scm_substring_hash_ci 150
scm_substring_move_x 147
scm_substring_read_only 145
scm_substring_shared 145
scm_substring_to_list 144
scm_substring_upcase 153
scm_substring_upcase_x 153
scm_sum 119
scm_supports_source_properties_p 433
SCM_SYMBOL 103
scm_symbol_fref 175
scm_symbol_fset_x 175
scm_symbol_hash 171
scm_symbol_interned_p 177
scm_symbol_p 172
scm_symbol_pref 175
scm_symbol_pset_x 175
scm_symbol_to_keyword 181
scm_symbol_to_string 172
scm_symlink 464
scm_sync 465
scm_sys_library_dir 413
scm_sys_make_void_port 319
scm_sys_package_data_dir 413
scm_sys_search_load_path 354
scm_sys_site_dir 413
scm_sys_string_dump 161
scm_syserror 302
scm_syserror_msg 302
scm_system 476
scm_system_async_mark 397
scm_system_async_mark_for_thread 397
scm_system_star 476
scm_take_c32vector 555
scm_take_c64vector 555
scm_take_f32vector 555
scm_take_f64vector 555
scm_take_locale_string 158
scm_take_locale_stringn 158
scm_take_locale_symbol 174
scm_take_locale_symboln 174
scm_take_s16vector 555
scm_take_s32vector 555
scm_take_s64vector 555
scm_take_s8vector 554
scm_take_u16vector 554
scm_take_u32vector 555
scm_take_u64vector 555
scm_take_u8vector 554
scm_tcgetpgrp 482
scm_tcsetpgrp 482
scm_textdomain 428
scm_thread_cleanup 400
scm_thread_exited_p 399
scm_thread_p 399
scm_throw 296
scm_thunk_p 245
scm_times 471
scm_tmpfile 466

scm_tmpnam ... 465
scm_to_bool ... 105
scm_to_char ... 108
scm_to_char_set ... 136
scm_to_double ... 112
scm_to_int ... 109
scm_to_int16 ... 109
scm_to_int32 ... 109
scm_to_int64 ... 109
scm_to_int8 ... 109
scm_to_intmax ... 109
scm_to_locale_string ... 159
scm_to_locale_stringbuf ... 159
scm_to_locale_stringn ... 159
scm_to_long ... 109
scm_to_long_long ... 109
scm_to_mpz ... 110
scm_to_pointer ... 390
scm_to_schar ... 108
scm_to_short ... 109
scm_to_signed_integer ... 108
scm_to_size_t ... 109
scm_to_sockaddr ... 493
scm_to_ssize_t ... 109
scm_to_uchar ... 108
scm_to_uint ... 109
scm_to_uint16 ... 109
scm_to_uint32 ... 109
scm_to_uint64 ... 109
scm_to_uint8 ... 109
scm_to_uintmax ... 109
scm_to_ulong ... 109
scm_to_ulong_long ... 109
scm_to_unsigned_integer ... 108
scm_to_ushort ... 109
scm_total_processor_count ... 478
scm_transpose_array ... 205
scm_truncate_divide ... 121
scm_truncate_file ... 310
scm_truncate_number ... 120
scm_truncate_quotient ... 121
scm_truncate_remainder ... 121
scm_try_arbiter ... 396
scm_try_mutex ... 402
scm_ttyname ... 482
scm_typed_array_p ... 200
scm_tzset ... 470
scm_u16vector ... 551
scm_u16vector_elements ... 555
scm_u16vector_length ... 552
scm_u16vector_p ... 550
scm_u16vector_ref ... 553
scm_u16vector_set_x ... 553
scm_u16vector_to_list ... 554
scm_u16vector_writable_elements ... 556
scm_u32vector ... 551
scm_u32vector_elements ... 555
scm_u32vector_length ... 552
scm_u32vector_p ... 550
scm_u32vector_ref ... 553
scm_u32vector_set_x ... 553
scm_u32vector_to_list ... 554
scm_u32vector_writable_elements ... 556
scm_u64vector ... 552
scm_u64vector_elements ... 555
scm_u64vector_length ... 552
scm_u64vector_p ... 550
scm_u64vector_ref ... 553
scm_u64vector_set_x ... 553
scm_u64vector_to_list ... 554
scm_u64vector_writable_elements ... 556
scm_u8_list_to_bytevector ... 166
scm_u8vector ... 551
scm_u8vector_elements ... 555
scm_u8vector_length ... 552
scm_u8vector_p ... 550
scm_u8vector_ref ... 552
scm_u8vector_set_x ... 553
scm_u8vector_to_list ... 554
scm_u8vector_writable_elements ... 556
scm_ucs_range_to_char_set ... 135
scm_ucs_range_to_char_set_x ... 136
scm_uint_list_to_bytevector ... 166
scm_umask ... 473
scm_uname ... 500
SCM_UNBNDP ... 736
SCM_UNDEFINED ... 735
scm_uniform_array_read_x ... 203
scm_uniform_array_write ... 203
scm_uniform_vector_elements ... 557
scm_uniform_vector_length ... 556
scm_uniform_vector_p ... 556
scm_uniform_vector_ref ... 557
scm_uniform_vector_set_x ... 557
scm_uniform_vector_to_list ... 557
scm_uniform_vector_writable_elements ... 557
scm_unlock_mutex ... 402
scm_unlock_mutex_timed ... 402
SCM_UNPACK ... 100
scm_unread_char ... 307, 456
scm_unread_string ... 307
SCM_UNSPECIFIED ... 735
scm_usleep ... 481
scm_utf16_to_string ... 168
scm_utf32_to_string ... 168
scm_utf8_to_string ... 168
scm_utime ... 463
scm_values ... 290
SCM_VARIABLE ... 103
scm_variable_bound_p ... 374
SCM_VARIABLE_INIT ... 103
scm_variable_p ... 374
scm_variable_ref ... 374
scm_variable_set_x ... 374
scm_variable_unset_x ... 374
scm_vector ... 191

scm_vector_copy ... 192
scm_vector_elements ... 193
scm_vector_fill_x ... 192
scm_vector_length ... 191
scm_vector_move_left_x ... 192
scm_vector_move_right_x ... 193
scm_vector_p ... 191
scm_vector_ref ... 192
scm_vector_set_x ... 192
scm_vector_to_list ... 191
scm_vector_writable_elements ... 194
scm_version ... 412
scm_wait_condition_variable ... 403
scm_waitpid ... 475
scm_weak_key_hash_table_p ... 363
scm_weak_value_hash_table_p ... 363
scm_weak_vector ... 363
scm_weak_vector_p ... 363
scm_with_continuation_barrier ... 304
scm_with_dynamic_state ... 407
scm_with_fluid ... 406
scm_with_fluids ... 406
scm_with_guile ... 101
scm_with_throw_handler ... 294
scm_without_guile ... 404
scm_write_char ... 309
scm_write_line ... 311
scm_write_objcode ... 770
scm_write_string_partial ... 313
scm_wrong_num_args ... 302
scm_wrong_type_arg ... 302
scm_wrong_type_arg_msg ... 302
scm_xsubstring ... 157
scm_zero_p ... 117
SCM2PTR ... 736
script-stexi-documentation ... 683
sdocbook-flatten ... 678
search-path ... 355
second ... 536
seconds->time ... 567
seed->random-state ... 128
seek ... 310
select ... 460
select-kids ... 675
send ... 497
sendto ... 498
serious-condition? ... 580, 609
serve-one-client ... 524
servent:aliases ... 491
servent:name ... 491
servent:port ... 491
servent:proto ... 491
set! ... 594, 690
set-buffered-input-continuation?! ... 653
set-car! ... 184, 755
set-cdr! ... 184, 755
set-current-dynamic-state ... 407
set-current-error-port ... 314
set-current-input-port ... 314
set-current-module ... 375
set-current-output-port ... 314
set-object-properties! ... 268
set-object-property! ... 268
set-port-column! ... 308
set-port-conversion-strategy! ... 306
set-port-encoding! ... 305, 356
set-port-filename! ... 316
set-port-line! ... 308
set-port-position! ... 325
set-port-revealed! ... 454
set-procedure-properties! ... 246
set-procedure-property! ... 246
set-program-arguments ... 471
set-readline-input-port! ... 630
set-readline-output-port! ... 630
set-readline-prompt! ... 630
set-record-type-printer! ... 560
set-source-properties! ... 433
set-source-property! ... 433
set-struct-vtable-name! ... 217
set-symbol-property! ... 175
set-thread-cleanup! ... 400
set-time-nanosecond! ... 569
set-time-second! ... 569
set-time-type! ... 569
set-tm:gmtoff ... 469
set-tm:hour ... 468
set-tm:isdst ... 469
set-tm:mday ... 469
set-tm:min ... 468
set-tm:mon ... 469
set-tm:sec ... 468
set-tm:wday ... 469
set-tm:yday ... 469
set-tm:year ... 469
set-tm:zone ... 469
set-vm-trace-level! ... 441
setaffinity ... 478
setegid ... 475
setenv ... 472
seteuid ... 474
setgid ... 474
setgr ... 468
setgrent ... 467
setgroups ... 474
sethost ... 489
sethostent ... 489
sethostname ... 501
setitimer ... 481
setlocale ... 501
setnet ... 490
setnetent ... 490
setpgid ... 475
setpriority ... 478
setproto ... 491
setprotoent ... 490

setpw ... 467
setpwent ... 467
setserv ... 492
setservent ... 492
setsid ... 475
setsockopt ... 494
setter ... 247
setuid ... 474
setvbuf ... 458
seventh ... 536
shallow-clone ... 708
shared-array-increments ... 205
shared-array-offset ... 205
shared-array-root ... 205
shift ... 288
show ... 47
shutdown ... 496
sigaction ... 480
signal-condition-variable ... 403
simple-conditions ... 609
simple-format ... 309
sin ... 123, 595
sinh ... 124
sint-list->bytevector ... 167
sixth ... 536
sizeof ... 392
skip-until ... 672
skip-while ... 672
sleep ... 481
sloppy-assoc ... 225
sloppy-assq ... 224
sloppy-assv ... 225
slot-bound-using-class? ... 706
slot-bound? ... 706
slot-definition-accessor ... 704
slot-definition-allocation ... 704
slot-definition-getter ... 704
slot-definition-init-form ... 704
slot-definition-init-keyword ... 704
slot-definition-init-thunk ... 704
slot-definition-init-value ... 704
slot-definition-name ... 704
slot-definition-options ... 704
slot-definition-setter ... 704
slot-exists-using-class? ... 706
slot-exists? ... 706
slot-init-function ... 704
slot-missing ... 707
slot-ref ... 687, 706, 756
slot-ref-using-class ... 706
slot-set ... 756
slot-set! ... 687, 706
slot-set-using-class! ... 707
slot-unbound ... 707
sockaddr:addr ... 493
sockaddr:fam ... 492
sockaddr:flowinfo ... 493
sockaddr:path ... 492
sockaddr:port ... 493
sockaddr:scopeid ... 493
socket ... 494
socketpair ... 494
sort ... 269
sort! ... 269
sort-list ... 269
sort-list! ... 269
sorted? ... 269
source-properties ... 433
source-property ... 433
source:addr ... 239
source:column ... 239
source:file ... 239
source:line ... 239
span ... 542
span! ... 542
split-and-decode-uri-path ... 506
split-at ... 537
split-at! ... 537
sqrt ... 123, 595
SRV:send-reply ... 673
ssax:complete-start-tag ... 672
ssax:make-elem-parser ... 672
ssax:make-parser ... 672
ssax:make-pi-parser ... 672
ssax:read-attributes ... 672
ssax:read-cdata-body ... 671
ssax:read-char-data ... 672
ssax:read-char-ref ... 672
ssax:read-external-id ... 672
ssax:read-markup-token ... 671
ssax:read-pi-body-as-string ... 671
ssax:reverse-collect-str-drop-ws ... 671
ssax:skip-internal-dtd ... 671
ssax:uri-string->symbol ... 671
ssax:xml->sxml ... 672
stable-sort ... 269
stable-sort! ... 269
stack-id ... 431
stack-length ... 431
stack-ref ... 431
stack? ... 431
standard-error-port ... 330
standard-input-port ... 326
standard-output-port ... 330
start-stack ... 431
stat ... 461
stat:atime ... 462
stat:atimensec ... 462
stat:blksize ... 462
stat:blocks ... 462
stat:ctime ... 462
stat:ctimensec ... 462
stat:dev ... 461
stat:gid ... 461
stat:ino ... 461
stat:mode ... 461

stat:mtime ... 462
stat:mtimensec ... 462
stat:nlink ... 461
stat:perms ... 462
stat:rdev ... 461
stat:size ... 461
stat:type ... 462
stat:uid ... 461
statistics ... 50
statprof ... 666
statprof-accumulated-time ... 664
statprof-active? ... 664
statprof-call-data->stats ... 665
statprof-call-data-calls ... 665
statprof-call-data-cum-samples ... 665
statprof-call-data-name ... 665
statprof-call-data-self-samples ... 665
statprof-display ... 665
statprof-display-anomolies ... 665
statprof-fetch-call-tree ... 665
statprof-fetch-stacks ... 665
statprof-fold-call-data ... 665
statprof-proc-call-data ... 665
statprof-reset ... 664
statprof-sample-count ... 664
statprof-start ... 664
statprof-stats-%-time-in-proc ... 665
statprof-stats-calls ... 665
statprof-stats-cum-secs-in-proc ... 665
statprof-stats-cum-secs-per-call ... 665
statprof-stats-proc-name ... 665
statprof-stats-self-secs-in-proc ... 665
statprof-stats-self-secs-per-call ... 665
statprof-stop ... 664
status:exit-val ... 476
status:stop-sig ... 476
status:term-sig ... 476
step ... 50
stexi->plain-text ... 682
stexi->shtml ... 679
stexi->sxml ... 677
stexi->texi ... 683
stexi-extract-index ... 679
stream->list ... 652
stream->list&length ... 652
stream->reversed-list ... 652
stream->reversed-list&length ... 652
stream->vector ... 652
stream-car ... 651
stream-cdr ... 651
stream-fold ... 652
stream-for-each ... 652
stream-map ... 652
stream-null? ... 651
strerror ... 297
strftime ... 470
string ... 143, 597
string->bytevector ... 324
string->char-set ... 135
string->char-set! ... 135
string->date ... 574
string->header ... 506
string->keyword ... 591
string->list ... 144, 597
string->number ... 118, 597
string->pointer ... 390
string->symbol ... 173, 593
string->uri ... 505
string->utf16 ... 167
string->utf32 ... 167
string->utf8 ... 167
string->wrapped-lines ... 682
string-any ... 142
string-append ... 154, 597
string-append/shared ... 155
string-bytes-per-char ... 161
string-capitalize ... 154
string-capitalize! ... 154
string-ci->symbol ... 173
string-ci-hash ... 590, 623
string-ci< ... 149
string-ci<= ... 150
string-ci<=? ... 148, 601
string-ci<> ... 149
string-ci<? ... 148, 601
string-ci= ... 149
string-ci=? ... 148, 601
string-ci> ... 149
string-ci>= ... 150
string-ci>=? ... 148, 601
string-ci>? ... 148, 601
string-compare ... 148
string-compare-ci ... 149
string-concatenate ... 155
string-concatenate-reverse ... 155
string-concatenate-reverse/shared ... 155
string-concatenate/shared ... 155
string-contains ... 153
string-contains-ci ... 153
string-copy ... 145, 597
string-copy! ... 147
string-count ... 153
string-delete ... 157
string-downcase ... 153, 600
string-downcase! ... 154
string-drop ... 145
string-drop-right ... 146
string-every ... 142
string-fill! ... 147
string-filter ... 157
string-fold ... 156
string-fold-right ... 156
string-foldcase ... 600
string-for-each ... 155, 597
string-for-each-index ... 155
string-hash ... 150, 590, 623

string-hash-ci ... 150
string-index ... 151
string-index-right ... 152
string-join ... 143
string-length ... 144, 597
string-locale-ci<? ... 422
string-locale-ci=? ... 422
string-locale-ci>? ... 422
string-locale-downcase ... 423
string-locale-titlecase ... 423
string-locale-upcase ... 423
string-locale<? ... 422
string-locale>? ... 422
string-map ... 155
string-map! ... 155
string-match ... 336
string-normalize-nfc ... 151, 601
string-normalize-nfd ... 151, 601
string-normalize-nfkc ... 151, 601
string-normalize-nfkd ... 151, 601
string-null? ... 142
string-pad ... 146
string-pad-right ... 146
string-prefix-ci? ... 152
string-prefix-length ... 151
string-prefix-length-ci ... 151
string-prefix? ... 152
string-ref ... 144, 597
string-replace ... 157
string-reverse ... 154
string-reverse! ... 154
string-rindex ... 151
string-set! ... 146
string-skip ... 152
string-skip-right ... 152
string-split ... 144
string-suffix-ci? ... 152
string-suffix-length ... 151
string-suffix-length-ci ... 152
string-suffix? ... 152
string-tabulate ... 143
string-take ... 145
string-take-right ... 145
string-titlecase ... 154, 600
string-titlecase! ... 154
string-tokenize ... 157
string-trim ... 146
string-trim-both ... 146
string-trim-right ... 146
string-unfold ... 156
string-unfold-right ... 156
string-upcase ... 153, 600
string-upcase! ... 153
string-xcopy! ... 157
string< ... 149
string<= ... 149
string<=? ... 148, 597
string<> ... 149
string<? ... 147, 597
string= ... 149
string=? ... 147, 597
string> ... 149
string>= ... 149
string>=? ... 148, 597
string>? ... 148, 597
string? ... 142, 594
strptime ... 470
strtod ... 423, 424
struct-ref ... 216, 756
struct-set ... 756
struct-set! ... 216
struct-vtable ... 216, 756
struct-vtable-name ... 217
struct-vtable? ... 216
struct? ... 216, 756
sub ... 756
sub1 ... 756
subr-call ... 750
substring ... 145, 597
substring-fill! ... 147
substring-move! ... 147
substring/copy ... 145
substring/read-only ... 145
substring/shared ... 145
subtract-duration ... 570
subtract-duration! ... 570
supports-source-properties? ... 433
sxml->string ... 669
sxml->xml ... 669
sxml-match ... 656
sxml-match-let ... 661
sxml-match-let* ... 661
sxpath ... 676
symbol ... 172
symbol->keyword ... 181
symbol->string ... 172, 593
symbol-append ... 173
symbol-fref ... 175
symbol-fset! ... 175
symbol-hash ... 171, 623
symbol-interned? ... 177
symbol-pref ... 175
symbol-prefix-proc ... 366
symbol-property ... 175
symbol-property-remove! ... 176
symbol-pset! ... 175
symbol=? ... 595
symbol? ... 172, 593
symlink ... 464
sync ... 465
sync-q! ... 650
syntax ... 254, 620
syntax->datum ... 256, 621
syntax-case ... 253, 620
syntax-local-binding ... 258
syntax-locally-bound-identifiers ... 259

syntax-module ... 258
syntax-parameterize ... 263
syntax-rules ... 249, 594
syntax-source ... 258
syntax-violation ... 621
syntax-violation-form ... 610
syntax-violation-subform ... 610
syntax-violation? ... 610
system ... 476
system* ... 476
system-async-mark ... 397
system-error-errno ... 453

T

tail-apply ... 746
tail-call ... 746
tail-call/cc ... 748
tail-call/nargs ... 747
take ... 537
take! ... 537
take-after ... 675
take-right ... 537
take-until ... 675
take-while ... 542
take-while! ... 542
tan ... 124, 596
tanh ... 124
tcgetpgrp ... 482
tcsetpgrp ... 482
tenth ... 537
terminated-thread-exception? ... 567
texi->stexi ... 677
texi-command-depth ... 677
texi-fragment->stexi ... 677
textdomain ... 428
textual-port? ... 324
the-environment ... 357
third ... 536
thread-cleanup ... 400
thread-exited? ... 399
thread-join! ... 564
thread-name ... 564
thread-sleep! ... 564
thread-specific ... 564
thread-specific-set! ... 564
thread-start! ... 564
thread-terminate! ... 564
thread-yield! ... 564
thread? ... 399, 563
throw ... 296
thunk? ... 245
time ... 49
time->seconds ... 567
time-difference ... 570
time-difference! ... 570
time-monotonic->date ... 571
time-monotonic->time-tai ... 571
time-monotonic->time-tai! ... 571
time-monotonic->time-utc ... 571
time-monotonic->time-utc! ... 571
time-nanosecond ... 569
time-resolution ... 569
time-second ... 569
time-tai->date ... 572
time-tai->julian-day ... 572
time-tai->modified-julian-day ... 572
time-tai->time-monotonic ... 572
time-tai->time-monotonic! ... 572
time-tai->time-utc ... 572
time-tai->time-utc! ... 572
time-type ... 569
time-utc->date ... 572
time-utc->julian-day ... 572
time-utc->modified-julian-day ... 572
time-utc->time-monotonic ... 572
time-utc->time-monotonic! ... 572
time-utc->time-tai ... 572
time-utc->time-tai! ... 572
time<=? ... 569
time<? ... 569
time=? ... 569
time>=? ... 570
time>? ... 570
time? ... 566, 569
times ... 471
tm:gmtoff ... 469
tm:hour ... 468
tm:isdst ... 469
tm:mday ... 469
tm:min ... 468
tm:mon ... 469
tm:sec ... 468
tm:wday ... 469
tm:yday ... 469
tm:year ... 469
tm:zone ... 469
tmpfile ... 466
tmpnam ... 465
tms:clock ... 471
tms:cstime ... 471
tms:cutime ... 471
tms:stime ... 471
tms:utime ... 471
toplevel-ref ... 745
toplevel-set ... 745
total-processor-count ... 478
touch ... 410
trace ... 49
trace-calls-in-procedure ... 445
trace-calls-to-procedure ... 445
trace-instructions-in-procedure ... 446
tracepoint ... 50
transcoded-port ... 325
transcoder-codec ... 323
transcoder-eol-style ... 323

transcoder-error-handling-mode 323
transform-string 680
transpose-array 205
trap-at-procedure-call 443
trap-at-procedure-ip-in-range 443
trap-at-source-location 444
trap-calls-in-dynamic-extent 444
trap-calls-to-procedure 444
trap-enabled? 446
trap-frame-finish 444
trap-in-dynamic-extent 444
trap-in-procedure 443
trap-instructions-in-dynamic-extent 444
trap-instructions-in-procedure 443
trap-matching-instructions 444
trap-name 446
truncate 120, 310, 597
truncate-file 310
truncate-quotient 121
truncate-remainder 121
truncate-values 747
truncate/ 121
truncated-print 633
try-arbiter 396
try-mutex 402
ttyname 482
typed-array? 200
tzset 470

U

u16vector 551
u16vector->list 553
u16vector-length 552
u16vector-ref 552
u16vector-set! 553
u16vector? 550
u32vector 551
u32vector->list 553
u32vector-length 552
u32vector-ref 552
u32vector-set! 553
u32vector? 550
u64vector 551
u64vector->list 553
u64vector-length 552
u64vector-ref 552
u64vector-set! 553
u64vector? 550
u8-list->bytevector 166
u8vector 551
u8vector->list 553
u8vector-length 552
u8vector-ref 552
u8vector-set! 553
u8vector? 550
ucs-range->char-set 135
ucs-range->char-set! 136
uint-list->bytevector 166
umask 473
uname 500
uncaught-exception-reason 567
uncaught-exception? 567
undefined-violation? 611
unfold 540
unfold-right 540
uniform-array-read! 203
uniform-array-write 203
uniform-vector->list 557
uniform-vector-length 556
uniform-vector-ref 557
uniform-vector-set! 557
uniform-vector? 556
unless 281, 603
unlink 463
unlock-mutex 402
unquote 344, 595
unquote-splicing 344, 595
unread-char 307, 456
unread-string 307, 456
unsetenv 472
unsyntax 620
unsyntax-splicing 620
unwind 753
unwind-fluids 754
unzip1 538
unzip2 538
unzip3 538
unzip4 538
unzip5 538
up 49
update-direct-method! 722
update-direct-subclass! 722
update-instance-for-different-class 723
uri->string 505
uri-decode 505
uri-encode 505
uri-fragment 505
uri-host 505
uri-path 505
uri-port 505
uri-query 505
uri-scheme 505
uri-userinfo 505
uri? 505
urlify 679
use-modules 366
usleep 481
utf-16-codec 321
utf-8-codec 321
utf16->string 168
utf32->string 168
utf8->string 168
utime 463
utsname:machine 501
utsname:nodename 500

utsname:release ... 500
utsname:sysname ... 500
utsname:version ... 501

V

valid-header? ... 507
value-history-enabled? ... 46
values ... 290, 599
variable-bound? ... 374, 746
variable-ref ... 374, 746
variable-set ... 746
variable-set! ... 374
variable-unset! ... 374
variable? ... 374
vector ... 191, 599, 752
vector->list ... 191, 599
vector->stream ... 651
vector-copy ... 192
vector-fill! ... 192, 599
vector-for-each ... 599
vector-length ... 191, 599
vector-map ... 599
vector-move-left! ... 192
vector-move-right! ... 193
vector-ref ... 192, 599, 755
vector-set ... 755
vector-set! ... 192, 599
vector-sort ... 602
vector-sort! ... 602
vector? ... 191, 599
version ... 412
vhash-assoc ... 227
vhash-assq ... 227
vhash-assv ... 227
vhash-cons ... 226
vhash-consq ... 226
vhash-consv ... 226
vhash-delete ... 227
vhash-delq ... 227
vhash-delv ... 227
vhash-fold ... 227
vhash-fold* ... 227
vhash-fold-right ... 227
vhash-foldq* ... 227
vhash-foldv* ... 227
vhash? ... 226
violation? ... 610
vlist->list ... 212
vlist-append ... 212
vlist-cons ... 211
vlist-delete ... 212
vlist-drop ... 212
vlist-filter ... 212
vlist-fold ... 211
vlist-fold-right ... 211
vlist-for-each ... 212
vlist-head ... 211
vlist-length ... 212
vlist-map ... 212
vlist-null? ... 211
vlist-ref ... 212
vlist-reverse ... 212
vlist-tail ... 211
vlist-take ... 212
vlist-unfold ... 212
vlist-unfold-right ... 212
vlist? ... 211
vm-abort-continuation-hook ... 441
vm-apply-hook ... 441
vm-next-hook ... 441
vm-pop-continuation-hook ... 441
vm-push-continuation-hook ... 441
vm-restore-continuation-hook ... 441
vm-trace-level ... 441
void ... 755

W

wait-condition-variable ... 403
waitpid ... 475
warning? ... 609
weak-key-hash-table? ... 363
weak-value-hash-table? ... 363
weak-vector ... 363
weak-vector? ... 363
when ... 281, 603
while ... 284
who-condition? ... 610
width ... 50
wind ... 753
wind-fluids ... 754
with-code-coverage ... 448
with-continuation-barrier ... 304
with-default-trap-handler ... 447
with-dynamic-state ... 407
with-error-to-file ... 316
with-exception-handler ... 567, 607
with-fluid* ... 406
with-fluids ... 407
with-fluids* ... 406
with-input-from-file ... 316, 612
with-input-from-string ... 317
with-mutex ... 403
with-output-to-file ... 316, 613
with-output-to-string ... 317
with-parameters* ... 584
with-readline-completion-function ... 631
with-ssax-error-to-port ... 671
with-statprof ... 666
with-syntax ... 257
with-throw-handler ... 294
write ... 347, 613, 709
write-char ... 309, 613
write-client ... 524
write-header ... 508

write-headers 508
write-line 311
write-objcode 770
write-request 518
write-request-body 518
write-request-line 508
write-response 520
write-response-body 521
write-response-line 508
write-string/partial 313
write-with-shared-structure 582

X

xcons 535
xml->sxml 669
xml-token-head 671
xml-token-kind 671
xml-token? 671
xsubstring 157

Y

yield 399

Z

zero? 117, 596
zip 538

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.

%

%auto-compilation-options 352
%default-port-encoding 305
%global-locale 422
%guile-build-info 413
%host-type 414
%load-compiled-path 354
%load-extensions 354
%load-hook 353
%load-path 353
%null-pointer 389

&

&condition 580
&error 581
&message 580
&serious 580

*

features 414
random-state 128
sdocbook->stexi-rules 677
sdocbook-block-commands 677

_

_IOFBF 458
_IOLBF 458
_IONBF 458

A

accept 512
accept-charset 512
accept-encoding 513
accept-language 513
accept-ranges 515
after-gc-hook 275
age 515
allow 511
authorization 513

B

block-growth-factor 211

C

cache-control 510
char-set:ascii 140
char-set:blank 139
char-set:designated 140
char-set:digit 139
char-set:empty 140
char-set:full 140
char-set:graphic 139
char-set:hex-digit 140
char-set:iso-control 139
char-set:letter 139
char-set:letter+digit 139
char-set:lower-case 138
char-set:printing 139
char-set:punctuation 139
char-set:symbol 139
char-set:title-case 139
char-set:upper-case 139
char-set:whitespace 139
connection 510
content-encoding 511
content-language 511
content-length 511
content-location 511
content-md5 512
content-range 512
content-type 512
current-reader 353

D

date 510
double 388

E

etag 515
expect 513
expires 512

F

F_DUPFD 458
F_GETFD 459
F_GETFL 459
F_GETOWN 459
F_OK 460

F_SETFD ... 459
F_SETFL ... 459
F_SETOWN ... 459
FD_CLOEXEC ... 459
float ... 388
from ... 513

G

GUILE_AUTO_COMPILE ... 38
GUILE_HISTORY ... 39
GUILE_LOAD_COMPILED_PATH ... 39
GUILE_LOAD_PATH ... 39
GUILE_WARN_DEPRECATED ... 39

H

HOME ... 39
host ... 513

I

if-match ... 513
if-modified-since ... 514
if-none-match ... 514
if-range ... 514
if-unmodified-since ... 514
INADDR_ANY ... 484
INADDR_BROADCAST ... 484
INADDR_LOOPBACK ... 484
int ... 388
int16 ... 388
int32 ... 388
int64 ... 388
int8 ... 388
internal-time-units-per-second ... 471
IP_ADD_MEMBERSHIP ... 495
IP_DROP_MEMBERSHIP ... 495
IP_MULTICAST_IF ... 495
IP_MULTICAST_TTL ... 495
IPPROTO_IP ... 495
IPPROTO_TCP ... 495
IPPROTO_UDP ... 495
ITIMER_PROF ... 481
ITIMER_REAL ... 481
ITIMER_VIRTUAL ... 481

L

last-modified ... 512
LC_ALL ... 501
LC_COLLATE ... 501
LC_CTYPE ... 501
LC_MESSAGES ... 501
LC_MONETARY ... 501
LC_NUMERIC ... 501
LC_TIME ... 501
location ... 515
LOCK_EX ... 459
LOCK_NB ... 459
LOCK_SH ... 459
LOCK_UN ... 459
long ... 388
LTDL_LIBRARY_PATH ... 40

M

max-forwards ... 514
MSG_DONTROUTE ... 497, 498
MSG_OOB ... 497, 498
MSG_PEEK ... 497, 498

O

O_APPEND ... 456
O_CREAT ... 456
O_RDONLY ... 455
O_RDWR ... 456
O_WRONLY ... 456
OPEN_BOTH ... 483
OPEN_READ ... 483
OPEN_WRITE ... 483

P

PF_INET ... 494
PF_INET6 ... 494
PF_UNIX ... 494
PIPE_BUF ... 457
pragma ... 510
PRIO_PGRP ... 478
PRIO_PROCESS ... 478
PRIO_USER ... 478
proxy-authenticate ... 515
proxy-authorization ... 514

R

R_OK ... 460
range ... 514
referer ... 515
regexp/basic ... 337
regexp/extended ... 337
regexp/icase ... 337
regexp/newline ... 337
regexp/notbol ... 337
regexp/noteol ... 338
retry-after ... 516

S

SA_NOCLDSTOP ... 480
SA_RESTART ... 480
scm_after_gc_c_hook ... 275
scm_after_gc_hook ... 275

scm_after_sweep_c_hook 275
scm_before_gc_c_hook 274
scm_before_mark_c_hook 275
scm_before_sweep_c_hook 275
SCM_BOOL_F 105
SCM_BOOL_T 105
SCM_C_HOOK_AND 273
SCM_C_HOOK_NORMAL 273
SCM_C_HOOK_OR 273
scm_char_set_ascii 140
scm_char_set_blank 139
scm_char_set_designated 140
scm_char_set_digit 139
scm_char_set_empty 140
scm_char_set_full 140
scm_char_set_graphic 139
scm_char_set_hex_digit 140
scm_char_set_iso_control 139
scm_char_set_letter 139
scm_char_set_letter_and_digit 139
scm_char_set_lower_case 138
scm_char_set_printing 139
scm_char_set_punctuation 139
scm_char_set_symbol 139
scm_char_set_title_case 139
scm_char_set_upper_case 139
scm_char_set_whitespace 139
scm_endianness_big 162
scm_endianness_little 162
SCM_F_WIND_EXPLICITLY 301
scm_global_locale 422
scm_ptobs 332
scm_t_int16 108
scm_t_int32 108
scm_t_int64 108
scm_t_int8 108
scm_t_intmax 108
scm_t_uint16 108
scm_t_uint32 108
scm_t_uint64 108
scm_t_uint8 108
scm_t_uintmax 108
scm_vtable_index_layout 216
scm_vtable_index_printer 217
scm_vtable_index_vtable 217
scm_vtable_offset_user 218
SEEK_CUR 310
SEEK_END 310
SEEK_SET 310
server 516
SIGHUP 479
SIGINT 479
size_t 388
SO_BROADCAST 495
SO_DEBUG 495
SO_DONTROUTE 495
SO_ERROR 495
SO_KEEPALIVE 495
SO_LINGER 495
SO_NO_CHECK 495
SO_OOBINLINE 495
SO_PRIORITY 495
SO_RCVBUF 495
SO_REUSEADDR 495
SO_SNDBUF 495
SO_STYLE 495
SO_TYPE 495
SOCK_DGRAM 494
SOCK_RAW 494
SOCK_RDM 494
SOCK_SEQPACKET 494
SOCK_STREAM 494
SOL_SOCKET 495

T

te 515
texi-command-specs 677
time-duration 569
time-monotonic 569
time-process 569
time-tai 568
time-thread 569
time-utc 568
trailer 510
transfer-encoding 510

U

uint16 388
uint32 388
uint64 388
uint8 388
unsigned-int 388
unsigned-long 388
upgrade 510
user-agent 515

V

vary 516
via 511
vlist-null 211
void 388
vtable-index-layout 216
vtable-index-printer 217
vtable-index-vtable 217
vtable-offset-user 218

W

W_OK 460
WAIT_ANY 475
WAIT_MYPGRP 475
warning 511
WNOHANG 476

WUNTRACED 476
www-authenticate 516

X

X_OK 460

Type Index

This is an alphabetical list of all the important data types defined in the Guile Programmers Manual.

(

(@ ... 762
(@@ ... 762
(abort ... 765
(apply ... 763
(begin ... 763
(const ... 762
(define ... 763
(dynlet ... 764
(dynref ... 764
(dynset ... 764
(dynwind ... 764
(fix ... 765
(if ... 763
(lambda ... 763
(lambda-case ... 763
(let ... 764
(let-values ... 765
(letrec ... 764
(letrec* ... 764
(lexical ... 762
(primitive ... 762
(prompt ... 764
(set! ... 762, 763
(toplevel ... 763
(void) ... 762

<

<abort> ... 765
<application> ... 763
<conditional> ... 763
<const> ... 762
<dynlet> ... 764
<dynref> ... 764
<dynset> ... 764
<dynwind> ... 764
<fix> ... 765
<glil-bind> ... 766
<glil-branch> ... 767
<glil-call> ... 767
<glil-const> ... 766
<glil-kw-prelude> ... 766
<glil-label> ... 767
<glil-lexical> ... 767
<glil-module> ... 767
<glil-mv-bind> ... 766
<glil-mv-call> ... 767
<glil-opt-prelude> ... 766
<glil-program> ... 765
<glil-prompt> ... 767
<glil-source> ... 766
<glil-std-prelude> ... 765
<glil-toplevel> ... 767
<glil-unbind> ... 766
<glil-void> ... 766
<lambda-case> ... 763
<lambda> ... 763
<let-values> ... 765
<let> ... 764
<letrec> ... 764
<lexical-ref> ... 762
<lexical-set> ... 762
<module-ref> ... 762
<module-set> ... 762
<primitive-ref> ... 762
<prompt> ... 764
<sequence> ... 763
<toplevel-define> ... 763
<toplevel-ref> ... 763
<toplevel-set> ... 763
<void> ... 762

A

Alist ... 220
Arrays ... 198
Association Lists ... 220

B

Booleans ... 104

C

Characters ... 129
Complex numbers ... 112

D

Date ... 509

E

ETag ... 509
Exact numbers ... 113

H

Hash Tables ... 228
Hooks ... 270

I

Inexact numbers 113
Integer numbers 106

K

Keywords 178
KVList 509

L

Lists 185

N

Numbers 105

P

Pairs 182
Parameter 408

Q

QList 509
Quality 509
Queues 649

R

Rational numbers 110
Real numbers 110
Regular expressions 335

S

SCM 100
scm_port 332
scm_ptob_descriptor 332
scm_t_array_dim 207
scm_t_array_handle 206
scm_t_bits 100
scm_t_c_hook 273
scm_t_c_hook_function 274
scm_t_c_hook_type 273
scm_t_catch_body 294
scm_t_catch_handler 294
scm_t_dynwind_flags 300
scm_t_signed_bits 100
scm_t_string_failed_conversion_handler .. 159
scm_t_wchar 133
scm_t_wind_flags 300
SList 509
sockaddr 493
Socket address 492
Strings 140
struct sockaddr 493
Structures 214
Symbols 168

V

Variables 373
Vectors 190

R5RS Index

*

* ... 119

+

+ ... 119

-

- ... 119

/

/ ... 119

A

abs ... 119
acos ... 124
angle ... 118
append ... 187
apply ... 349
asin ... 124
assoc ... 222
assq ... 222
assv ... 222
atan ... 124

B

boolean? ... 105

C

call-with-current-continuation ... 288
call-with-input-file ... 316
call-with-output-file ... 316
call-with-values ... 291
car ... 183
cdr ... 183
ceiling ... 119
ceiling-quotient ... 119
ceiling-remainder ... 119
ceiling/ ... 119
centered-quotient ... 119
centered-remainder ... 119
centered/ ... 119
char->integer ... 132
char-alphabetic? ... 131
char-ci<=? ... 131
char-ci<? ... 131
char-ci=? ... 131
char-ci>=? ... 131
char-ci>? ... 131
char-downcase ... 132
char-lower-case? ... 131
char-numeric? ... 131
char-ready? ... 306
char-titlecase ... 132
char-upcase ... 132
char-upper-case? ... 131
char-whitespace? ... 131
char<=? ... 130
char<? ... 130
char=? ... 130
char>=? ... 130
char>? ... 130
char? ... 130
close-input-port ... 309
close-output-port ... 309
complex? ... 112
cons ... 182
cos ... 123
current-input-port ... 313
current-output-port ... 313

D

delay ... 356
display ... 347
div ... 598
div-and-mod ... 598
div0 ... 598
div0-and-mod0 ... 598
dynamic-wind ... 299

E

eof-object? ... 306
eq? ... 265
equal? ... 266
eqv? ... 266
euclidean-quotient ... 119
euclidean-remainder ... 119
euclidean/ ... 119
eval ... 348
even? ... 115
exact->inexact ... 113
exact? ... 113
exp ... 124
expt ... 123

F

floor ... 119
floor-quotient ... 119
floor-remainder ... 119
floor/ ... 119

for-each 190
force 356

G

gcd 115

I

imag-part 118
inexact->exact 113
inexact? 113
input-port? 305
integer->char 132
integer? 106
interaction-environment 348

L

lcm 115
length 186
list 186
list->string 143
list->symbol 172
list->vector 191
list-ref 187
list-tail 187
list? 185
load 352
log 124

M

magnitude 118
make-polar 118
make-rectangular 118
make-string 143
make-vector 191
map 190
max 119
member 189
memq 189
memv 189
min 119
mod 598
mod0 598
modulo 115

N

negative? 117
newline 308
not 104
null? 185
number->string 117
number? 105

O

odd? 115
open-input-file 315
open-output-file 315
output-port? 305

P

pair? 182
peek-char 307
positive? 117
print 347
procedure? 245

Q

quotient 115

R

rational? 110
read 346
read-char 307
real-part 118
real? 110
remainder 115
reverse 188
round 119
round-quotient 119
round-remainder 119
round/ 119

S

set-car! 184
set-cdr! 184
sin 123
sqrt 123
string 143
string->list 144
string->number 117
string->symbol 173
string-append 154
string-ci<? 148
string-ci=? 148
string-ci>=? 148
string-ci>? 148
string-copy 145
string-fill! 147
string-length 144
string-ref 144
string-set! 146
string<=? 148
string<? 147
string=? 147
string>=? 148
string>? 148

string? 142
substring 145
symbol 172
symbol->string 172
symbol-append 173
symbol? 172

T

tan 124
truncate 119
truncate-quotient 119
truncate-remainder 119
truncate/ 119

V

values 290
vector 191
vector->list 191
vector-fill! 192
vector-length 191
vector-ref 192
vector-set! 192
vector? 191

W

with-input-from-file 316
with-output-to-file 316
write 347
write-char 309

Z

zero? 117