MIT/GNU Scheme Reference Manual

for release 12.1
2022-11-10

by Chris Hanson
the MIT Scheme Team
and a cast of thousands

This manual documents MIT/GNU Scheme 12.1.

Copyright © 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022 Massachusetts Institute of Technology

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License."

Short Contents

Acknowledgements 1
1 Overview 3
2 Special Forms 15
3 Equivalence Predicates 57
4 Numbers 63
5 Characters 109
6 Strings 117
7 Lists 139
8 Vectors 157
9 Bit Strings 163
10 Miscellaneous Datatypes 167
11 Associations 181
12 Procedures 215
13 Environments 223
14 Input/Output 227
15 Operating-System Interface 275
16 Error System 309
17 Graphics 333
18 Standards Support 345
A GNU Free Documentation License 361
B Binding Index 369
C Concept Index 393

Table of Contents

Acknowledgements **1**

1 Overview **3**

1.1 Notational Conventions 4
1.1.1 Errors 4
1.1.2 Examples 4
1.1.3 Entry Format 5
1.2 Scheme Concepts 6
1.2.1 Variable Bindings 6
1.2.2 Environment Concepts 6
1.2.3 Initial and Current Environments 7
1.2.4 Static Scoping 7
1.2.5 True and False 8
1.2.6 External Representations 8
1.2.7 Disjointness of Types 9
1.2.8 Storage Model 9
1.3 Lexical Conventions 9
1.3.1 Whitespace 9
1.3.2 Delimiters 10
1.3.3 Identifiers 10
1.3.4 Uppercase and Lowercase 10
1.3.5 Naming Conventions 10
1.3.6 Comments 11
1.3.7 Additional Notations 11
1.4 Expressions 12
1.4.1 Literal Expressions 13
1.4.2 Variable References 13
1.4.3 Special Form Syntax 13
1.4.4 Procedure Call Syntax 13

2 Special Forms **15**

2.1 Lambda Expressions 15
2.2 Lexical Binding 17
2.3 Dynamic Binding 20
2.3.1 Fluid-Let 22
2.4 Definitions 22
2.4.1 Top-Level Definitions 23
2.4.2 Internal Definitions 23
2.5 Assignments 24
2.6 Quoting 24
2.7 Conditionals 26
2.8 Sequencing 28

2.9 Iteration 29
2.10 Structure Definitions 31
2.11 Macros 36
2.11.1 Binding Constructs for Syntactic Keywords 37
2.11.2 Pattern Language 40
2.11.3 Syntactic Closures 42
2.11.3.1 Syntax Terminology 42
2.11.3.2 Transformer Definition 43
2.11.3.3 Identifiers 47
2.11.4 Explicit Renaming 49
2.12 SRFI syntax 51
2.12.1 cond-expand (SRFI 0) 52
2.12.2 receive (SRFI 8) 53
2.12.3 and-let* (SRFI 2) 54
2.12.4 define-record-type (SRFI 9) 54

3 Equivalence Predicates 57

4 Numbers 63
4.1 Numerical types 63
4.2 Exactness 64
4.3 Implementation restrictions 64
4.4 Syntax of numerical constants 66
4.5 Numerical operations 66
4.6 Numerical input and output 82
4.7 Bit operations 85
4.8 Fixnum and Flonum Operations 87
4.8.1 Fixnum Operations 88
4.8.2 Flonum Operations 90
4.8.3 Floating-Point Environment 98
4.8.4 Floating-Point Exceptions 100
4.8.5 Floating-Point Rounding Mode 103
4.9 Random Number Generation 104

5 Characters 109
5.1 Character implementation 112
5.2 Unicode 113
5.3 Character Sets 115

6 Strings 117
6.1 Searching and Matching Strings 129
6.2 Regular Expressions 132
6.2.1 Regular S-Expressions 132
6.2.2 Regsexp Procedures 137

7 Lists ... 139
7.1 Pairs ... 140
7.2 Construction of Lists ... 142
7.3 Selecting List Components ... 144
7.4 Cutting and Pasting Lists ... 145
7.5 Filtering Lists ... 147
7.6 Searching Lists ... 149
7.7 Mapping of Lists ... 150
7.8 Folding of Lists ... 152
7.9 Miscellaneous List Operations ... 154

8 Vectors ... 157
8.1 Construction of Vectors ... 157
8.2 Selecting Vector Components ... 159
8.3 Cutting Vectors ... 160
8.4 Modifying Vectors ... 160

9 Bit Strings ... 163
9.1 Construction of Bit Strings ... 163
9.2 Selecting Bit String Components ... 164
9.3 Cutting and Pasting Bit Strings ... 164
9.4 Bitwise Operations on Bit Strings ... 165
9.5 Modification of Bit Strings ... 165
9.6 Integer Conversions of Bit Strings ... 166

10 Miscellaneous Datatypes ... 167
10.1 Booleans ... 167
10.2 Symbols ... 168
10.3 Parameters ... 171
10.3.1 Cells ... 172
10.4 Records ... 172
10.5 Promises ... 174
10.6 Streams ... 176
10.7 Weak References ... 177
10.7.1 Weak Pairs ... 178
10.7.2 Ephemerons ... 179
10.7.3 Reference barriers ... 180

11 Associations ... 181
11.1 Association Lists ... 181
11.2 1D Tables ... 183
11.3 The Association Table ... 184
11.4 Hash Tables ... 185
11.4.1 Construction of Hash Tables ... 185
11.4.2 Basic Hash Table Operations ... 191
11.4.3 Resizing of Hash Tables ... 193
11.4.4 Address Hashing ... 195
11.5 Object Hashing ... 196
11.6 Red-Black Trees ... 197
11.7 Weight-Balanced Trees ... 201
11.7.1 Construction of Weight-Balanced Trees ... 202
11.7.2 Basic Operations on Weight-Balanced Trees ... 203
11.7.3 Advanced Operations on Weight-Balanced Trees ... 204
11.7.4 Indexing Operations on Weight-Balanced Trees ... 207
11.8 Associative Maps ... 208
11.8.1 Amap constructors ... 208
11.8.2 Amap predicates ... 210
11.8.3 Amap accessors ... 210
11.8.4 Amap mutators ... 211
11.8.5 Amap mapping and folding ... 212
11.8.6 Amap contents ... 212
11.8.7 Amap copying and conversion ... 213
11.8.8 Amaps as sets ... 213

12 Procedures ... 215
12.1 Procedure Operations ... 215
12.2 Arity ... 216
12.3 Primitive Procedures ... 217
12.4 Continuations ... 218
12.5 Application Hooks ... 220

13 Environments ... 223
13.1 Environment Operations ... 223
13.2 Environment Variables ... 225
13.3 REPL Environment ... 225
13.4 Top-level Environments ... 226

14 Input/Output 227

14.1 Ports 227
14.2 File Ports 229
14.3 String Ports 231
14.4 Bytevector Ports 233
14.5 Input Procedures 233
14.5.1 Reader Controls 237
14.6 Output Procedures 238
14.7 Blocking Mode 242
14.8 Terminal Mode 243
14.9 Format 244
14.10 Custom Output 246
14.11 Prompting 247
14.12 Textual Port Primitives 249
14.12.1 Textual Port Types 249
14.12.2 Constructors and Accessors for Textual Ports 250
14.12.3 Textual Input Port Operations 251
14.12.4 Textual Output Port Operations 252
14.13 Parser Buffers 254
14.14 Parser Language 257
14.14.1 *Matcher 259
14.14.2 *Parser 262
14.14.3 Parser-language Macros 265
14.15 XML Support 266
14.15.1 XML Input 266
14.15.2 XML Output 267
14.15.3 XML Names 268
14.15.4 XML Structure 270

15 Operating-System Interface 275

15.1 Pathnames 275
15.1.1 Filenames and Pathnames 276
15.1.2 Components of Pathnames 277
15.1.3 Operations on Pathnames 280
15.1.4 Miscellaneous Pathname Procedures 283
15.2 Working Directory 284
15.3 File Manipulation 285
15.4 Directory Reader 291
15.5 Date and Time 291
15.5.1 Universal Time 292
15.5.2 Decoded Time 292
15.5.3 File Time 294
15.5.4 Time-Format Conversion 295
15.5.5 External Representation of Time 298
15.6 Machine Time 298
15.7 Subprocesses 300

15.7.1 Subprocess Procedures ... 301
15.7.2 Subprocess Conditions ... 301
15.7.3 Subprocess Options ... 302
15.8 TCP Sockets ... 305
15.9 Miscellaneous OS Facilities ... 306

16 Error System ... 309
16.1 Condition Signalling ... 310
16.2 Error Messages ... 312
16.3 Condition Handling ... 313
16.4 Restarts ... 315
16.4.1 Establishing Restart Code ... 316
16.4.2 Invoking Standard Restart Code ... 317
16.4.3 Finding and Invoking General Restart Code ... 319
16.4.4 The Named Restart Abstraction ... 320
16.5 Condition Instances ... 320
16.5.1 Generating Operations on Conditions ... 321
16.5.2 Condition Abstraction ... 322
16.5.3 Simple Operations on Condition Instances ... 322
16.6 Condition Types ... 323
16.7 Condition-Type Taxonomy ... 324

17 Graphics ... 333
17.1 Opening and Closing of Graphics Devices ... 333
17.2 Coordinates for Graphics ... 334
17.3 Drawing Graphics ... 334
17.4 Characteristics of Graphics Output ... 336
17.5 Buffering of Graphics Output ... 337
17.6 Clipping of Graphics Output ... 338
17.7 Custom Graphics Operations ... 338
17.8 Images ... 338
17.9 X Graphics ... 339
17.9.1 X Graphics Type ... 340
17.9.2 Utilities for X Graphics ... 340
17.9.3 Custom Operations on X Graphics Devices ... 341

18 Standards Support ... 345
18.1 Revised^7 Report on the Algorithmic Language Scheme ... 345
18.2 SRFI 1: List Library ... 347
18.3 SRFI 2: `and-let*` ... 349
18.4 SRFI 8: `receive` ... 349
18.5 SRFI 9: Record Types ... 350
18.6 SRFI 14: Character-set Library ... 350
18.7 SRFI 23: Error Reporting Mechanism ... 351
18.8 SRFI 27: Sources of Random Bits ... 351

18.9 SRFI 39: Parameter Objects 352
18.10 SRFI 69: Basic Hash Tables 352
18.11 SRFI 112: Environment Inquiry 352
18.12 SRFI 115: Scheme Regular Expressions 353
18.13 SRFI 124: Ephemerons 353
18.14 SRFI 125: Intermediate Hash Tables 354
18.15 SRFI 128: Comparators (reduced) 355
18.16 SRFI 129: Titlecase procedures 355
18.17 SRFI 131: ERRR5RS Record Syntax (reduced) 356
18.18 SRFI 133: Vector Library (R7RS-compatible) 356
18.19 SRFI 140: Immutable Strings 357
18.20 SRFI 143: Fixnums 358
18.21 SRFI 158: Generators and Accumulators 359
18.22 SRFI 162: Comparators sublibrary 360
18.23 SRFI 219: Define higher-order lambda 360

Appendix A GNU Free Documentation License .. 361

A.1 ADDENDUM: How to use this License for your documents ... 367

Appendix B Binding Index 369

Appendix C Concept Index 393

Acknowledgements

While "a cast of thousands" may be an overstatement, it is certainly the case that this document represents the work of many people. First and foremost, thanks go to the authors of the *Revised^4 Report on the Algorithmic Language Scheme*, from which much of this document is derived. Thanks also to BBN Advanced Computers Inc. for the use of parts of their *Butterfly Scheme Reference*, and to Margaret O'Connell for translating it from BBN's text-formatting language to ours.

Special thanks to Richard Stallman, Bob Chassell, and Brian Fox, all of the Free Software Foundation, for creating and maintaining the Texinfo formatting language in which this document is written.

This report describes research done at the Artificial Intelligence Laboratory and the Laboratory for Computer Science, both of the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense and by the National Science Foundation.

1 Overview

This manual is a detailed description of the MIT/GNU Scheme runtime system. It is intended to be a reference document for programmers. It does not describe how to run Scheme or how to interact with it — that is the subject of the *MIT/GNU Scheme User's Manual*.

This chapter summarizes the semantics of Scheme, briefly describes the MIT/GNU Scheme programming environment, and explains the syntactic and lexical conventions of the language. Subsequent chapters describe special forms, numerous data abstractions, and facilities for input and output.

Throughout this manual, we will make frequent references to *standard Scheme*, which is the language defined by the document *Revised[7] Report on the Algorithmic Language Scheme*; Alex Shinn, John Cowan, and Arthur A. Gleckler (Editors). MIT/GNU Scheme is an extension of standard Scheme.

These are the significant semantic characteristics of the Scheme language:

Variables are statically scoped
: Scheme is a *statically scoped* programming language, which means that each use of a variable is associated with a lexically apparent binding of that variable. Algol is another statically scoped language.

Types are latent
: Scheme has *latent* types as opposed to *manifest* types, which means that Scheme associates types with values (or objects) rather than with variables. Other languages with latent types (also referred to as *weakly* typed or *dynamically* typed languages) include APL, Snobol, and other dialects of Lisp. Languages with manifest types (sometimes referred to as *strongly* typed or *statically* typed languages) include Algol 60, Pascal, and C.

Objects have unlimited extent
: All objects created during a Scheme computation, including procedures and continuations, have unlimited extent; no Scheme object is ever destroyed. The system doesn't run out of memory because the garbage collector reclaims the storage occupied by an object when the object cannot possibly be needed by a future computation. Other languages in which most objects have unlimited extent include APL and other Lisp dialects.

Proper tail recursion
: Scheme is *properly tail-recursive*, which means that iterative computation can occur in constant space, even if the iterative computation is described by a syntactically recursive procedure. With a tail-recursive implementation, you can express iteration using the ordinary procedure-call mechanics; special iteration expressions are provided only for syntactic convenience.

Procedures are objects
: Scheme procedures are objects, which means that you can create them dynamically, store them in data structures, return them as the results of other procedures, and so on. Other languages with such procedure objects include Common Lisp and ML.

Continuations are explicit
: In most other languages, continuations operate behind the scenes. In Scheme, continuations are objects; you can use continuations for implementing a variety of advanced control constructs, including non-local exits, backtracking, and coroutines.

Arguments are passed by value
: Arguments to Scheme procedures are passed by value, which means that Scheme evaluates the argument expressions before the procedure gains control, whether or not the procedure needs the result of the evaluations. ML, C, and APL are three other languages that pass arguments by value. In languages such as SASL and Algol 60, argument expressions are not evaluated unless the values are needed by the procedure.

Scheme uses a parenthesized-list Polish notation to describe programs and (other) data. The syntax of Scheme, like that of most Lisp dialects, provides for great expressive power, largely due to its simplicity. An important consequence of this simplicity is the susceptibility of Scheme programs and data to uniform treatment by other Scheme programs. As with other Lisp dialects, the `read` primitive parses its input; that is, it performs syntactic as well as lexical decomposition of what it reads.

1.1 Notational Conventions

This section details the notational conventions used throughout the rest of this document.

1.1.1 Errors

When this manual uses the phrase "an error will be signalled," it means that Scheme will call `error`, which normally halts execution of the program and prints an error message.

When this manual uses the phrase "it is an error," it means that the specified action is not valid in Scheme, but the system may or may not signal the error. When this manual says that something "must be," it means that violating the requirement is an error.

1.1.2 Examples

This manual gives many examples showing the evaluation of expressions. The examples have a common format that shows the expression being evaluated on the left hand side, an "arrow" in the middle, and the value of the expression written on the right. For example:

```
(+ 1 2)          ⇒  3
```

Sometimes the arrow and value will be moved under the expression, due to lack of space. Occasionally we will not care what the value is, in which case both the arrow and the value are omitted.

If an example shows an evaluation that results in an error, an error message is shown, prefaced by ' error ':

```
(+ 1 'foo)                          error  Illegal datum
```

An example that shows printed output marks it with ' ⊣ ':

```
(begin (write 'foo) 'bar)
     ⊣ foo
     ⇒ bar
```

When this manual indicates that the value returned by some expression is *unspecified*, it means that the expression will evaluate to some object without signalling an error, but that programs should not depend on the value in any way.

1.1.3 Entry Format

Each description of an MIT/GNU Scheme variable, special form, or procedure begins with one or more header lines in this format:

`template` [*category*]

where *category* specifies the kind of item ("variable", "special form", or "procedure"). The form of *template* is interpreted depending on *category*.

Variable *Template* consists of the variable's name.

Parameter *Template* consists of the parameter's name. See Section 2.3 [Dynamic Binding], page 20, and Section 10.3 [Parameters], page 171, for more information.

Special Form

Template starts with the syntactic keyword of the special form, followed by a description of the special form's syntax. The description is written using the following conventions.

Named components are italicized in the printed manual, and uppercase in the Info file. "Noise" keywords, such as the `else` keyword in the `cond` special form, are set in a fixed width font in the printed manual; in the Info file they are not distinguished. Parentheses indicate themselves.

A horizontal ellipsis (. . .) is describes repeated components. Specifically,

thing . . .

indicates *zero* or more occurrences of *thing*, while

thing thing . . .

indicates *one* or more occurrences of *thing*.

Brackets, `[ ]`, enclose optional components.

Several special forms (e.g. `lambda`) have an internal component consisting of a series of expressions; usually these expressions are evaluated sequentially under conditions that are specified in the description of the special form. This sequence of expressions is commonly referred to as the *body* of the special form.

Procedure *Template* starts with the name of the variable to which the procedure is bound, followed by a description of the procedure's arguments. The arguments are described using "lambda list" notation (see Section 2.1 [Lambda Expressions], page 15), except that brackets are used to denote optional arguments, and ellipses are used to denote "rest" arguments.

The names of the procedure's arguments are italicized in the printed manual, and uppercase in the Info file.

When an argument names a Scheme data type, it indicates that the argument must be that type of data object. For example,

`cdr` *pair* [procedure]
indicates that the standard Scheme procedure `cdr` takes one argument, which must be a pair.

Many procedures signal an error when an argument is of the wrong type; usually this error is a condition of type `condition-type:wrong-type-argument`.

In addition to the standard data-type names (*pair*, *list*, *boolean*, *string*, etc.), the following names as arguments also imply type restrictions:

- *object*: any object
- *thunk*: a procedure of no arguments
- *x*, *y*: real numbers
- *q*, *n*: integers
- *k*: an exact non-negative integer

Some examples:

`list` *object ...* [procedure]
indicates that the standard Scheme procedure `list` takes zero or more arguments, each of which may be any Scheme object.

`write-char` *char* [*output-port*] [procedure]
indicates that the standard Scheme procedure `write-char` must be called with a character, *char*, and may also be called with a character and an output port.

1.2 Scheme Concepts

1.2.1 Variable Bindings

Any identifier that is not a syntactic keyword may be used as a variable (see Section 1.3.3 [Identifiers], page 10). A variable may name a location where a value can be stored. A variable that does so is said to be *bound* to the location. The value stored in the location to which a variable is bound is called the variable's *value*. (The variable is sometimes said to *name* the value or to be *bound to* the value.)

A variable may be bound but still not have a value; such a variable is said to be *unassigned*. Referencing an unassigned variable is an error. When this error is signalled, it is a condition of type `condition-type:unassigned-variable`; sometimes the compiler does not generate code to signal the error. Unassigned variables are useful only in combination with side effects (see Section 2.5 [Assignments], page 24).

1.2.2 Environment Concepts

An *environment* is a set of variable bindings. If an environment has no binding for a variable, that variable is said to be *unbound* in that environment. Referencing an unbound variable signals a condition of type `condition-type:unbound-variable`.

A new environment can be created by *extending* an existing environment with a set of new bindings. Note that "extending an environment" does **not** modify the environment; rather, it creates a new environment that contains the new bindings and the old ones. The new bindings *shadow* the old ones; that is, if an environment that contains a binding for `x`

is extended with a new binding for `x`, then only the new binding is seen when `x` is looked up in the extended environment. Sometimes we say that the original environment is the *parent* of the new one, or that the new environment is a *child* of the old one, or that the new environment *inherits* the bindings in the old one.

Procedure calls extend an environment, as do `let`, `let*`, `letrec`, and `do` expressions. Internal definitions (see Section 2.4.2 [Internal Definitions], page 23) also extend an environment. (Actually, all the constructs that extend environments can be expressed in terms of procedure calls, so there is really just one fundamental mechanism for environment extension.) A top-level definition (see Section 2.4.1 [Top-Level Definitions], page 23) may add a binding to an existing environment.

1.2.3 Initial and Current Environments

MIT/GNU Scheme provides an *initial environment* that contains all of the variable bindings described in this manual. Most environments are ultimately extensions of this initial environment. In Scheme, the environment in which your programs execute is actually a child (extension) of the environment containing the system's bindings. Thus, system names are visible to your programs, but your names do not interfere with system programs.

The environment in effect at some point in a program is called the *current environment* at that point. In particular, every REP loop has a current environment. (REP stands for "read-eval-print"; the REP loop is the Scheme program that reads your input, evaluates it, and prints the result.) The environment of the top-level REP loop (the one you are in when Scheme starts up) starts as `user-initial-environment`, although it can be changed by the `ge` procedure. When a new REP loop is created, its environment is determined by the program that creates it.

1.2.4 Static Scoping

Scheme is a statically scoped language with block structure. In this respect, it is like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp.

The fact that Scheme is statically scoped (rather than dynamically bound) means that the environment that is extended (and becomes current) when a procedure is called is the environment in which the procedure was created (i.e. in which the procedure's defining lambda expression was evaluated), not the environment in which the procedure is called. Because all the other Scheme *binding expressions* can be expressed in terms of procedures, this determines how all bindings behave.

Consider the following definitions, made at the top-level REP loop (in the initial environment):

```
(define x 1)
(define (f x) (g 2))
(define (g y) (+ x y))
(f 5)                                    ⇒  3 ; not 7
```

Here `f` and `g` are bound to procedures created in the initial environment. Because Scheme is statically scoped, the call to `g` from `f` extends the initial environment (the one in which `g` was created) with a binding of `y` to `2`. In this extended environment, `y` is `2` and `x` is `1`. (In a dynamically bound Lisp, the call to `g` would extend the environment in effect during the call to `f`, in which `x` is bound to `5` by the call to `f`, and the answer would be `7`.)

Note that with static scoping, you can tell what binding a variable reference refers to just from looking at the text of the program; the referenced binding cannot depend on how the program is used. That is, the nesting of environments (their parent-child relationship) corresponds to the nesting of binding expressions in program text. (Because of this connection to the text of the program, static scoping is also called *lexical* scoping.) For each place where a variable is bound in a program there is a corresponding *region* of the program text within which the binding is effective. For example, the region of a binding established by a `lambda` expression is the entire body of the `lambda` expression. The documentation of each binding expression explains what the region of the bindings it makes is. A use of a variable (that is, a reference to or assignment of a variable) refers to the innermost binding of that variable whose region contains the variable use. If there is no such region, the use refers to the binding of the variable in the global environment (which is an ancestor of all other environments, and can be thought of as a region in which all your programs are contained).

1.2.5 True and False

In Scheme, the boolean values true and false are denoted by `#t` and `#f`. However, any Scheme value can be treated as a boolean for the purpose of a conditional test. This manual uses the word *true* to refer to any Scheme value that counts as true, and the word *false* to refer to any Scheme value that counts as false. In conditional tests, all values count as true except for `#f`, which counts as false (see Section 2.7 [Conditionals], page 26).

1.2.6 External Representations

An important concept in Scheme is that of the *external representation* of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters '`28`', and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters '`(8 13)`'.

The external representation of an object is not necessarily unique. The integer 28 also has representations '`#e28.000`' and '`#x1c`', and the list in the previous paragraph also has the representations '`( 08 13 )`' and '`(8 . (13 . ( )))`'.

Many objects have standard external representations, but some, such as procedures and circular data structures, do not have standard representations (although particular implementations may define representations for them).

An external representation may be written in a program to obtain the corresponding object (see Section 2.6 [Quoting], page 24).

External representations can also be used for input and output. The procedure `read` parses external representations, and the procedure `write` generates them. Together, they provide an elegant and powerful input/output facility.

Note that the sequence of characters '`(+ 2 6)`' is *not* an external representation of the integer 8, even though it *is* an expression that evaluates to the integer 8; rather, it is an external representation of a three-element list, the elements of which are the symbol `+` and the integers `2` and `6`. Scheme's syntax has the property that any sequence of characters that is an expression is also the external representation of some object. This can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates

writing programs such as interpreters and compilers that treat programs as data or data as programs.

1.2.7 Disjointness of Types

Every object satisfies at most one of the following predicates (but see Section 1.2.5 [True and False], page 8, for an exception):

```
bit-string?     environment?    port?           symbol?
boolean?        null?           procedure?      vector?
cell?           number?         promise?        weak-pair?
char?           pair?           string?
condition?
```

1.2.8 Storage Model

This section describes a model that can be used to understand Scheme's use of storage.

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the `string-set!` procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as `car`, `vector-ref`, or `string-ref`, is equivalent in the sense of `eqv?` to the object last stored in the location before the fetch.

Every location is marked to show whether it is in use. No variable or object ever refers to a location that is not in use. Whenever this document speaks of storage being allocated for a variable or object, what is meant is that an appropriate number of locations are chosen from the set of locations that are not in use, and the chosen locations are marked to indicate that they are now in use before the variable or object is made to denote them.

In many systems it is desirable for constants (i.e. the values of literal expressions) to reside in read-only memory. To express this, it is convenient to imagine that every object that denotes locations is associated with a flag telling whether that object is mutable or immutable. The constants and the strings returned by `symbol->string` are then the immutable objects, while all objects created by other procedures are mutable. It is an error to attempt to store a new value into a location that is denoted by an immutable object. Note that the MIT/GNU Scheme compiler takes advantage of this property to share constants, but that these constants are not immutable. Instead, two constants that are `equal?` may be `eq?` in compiled code.

1.3 Lexical Conventions

This section describes Scheme's lexical conventions.

1.3.1 Whitespace

Whitespace characters are spaces, newlines, tabs, and page breaks. Whitespace is used to improve the readability of your programs and to separate tokens from each other, when necessary. (A *token* is an indivisible lexical unit such as an identifier or number.) Whitespace

is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

1.3.2 Delimiters

All whitespace characters are *delimiters*. In addition, the following characters act as delimiters:

```
(  )  ;  "  '  `  |
```

Finally, these next characters act as delimiters, despite the fact that Scheme does not define any special meaning for them:

```
[  ]  {  }
```

For example, if the value of the variable `name` is `"max"`:

```
(list"Hi"name(+ 1 2))                    ⇒  ("Hi" "max" 3)
```

1.3.3 Identifiers

An *identifier* is a sequence of one or more non-delimiter characters. Identifiers are used in several ways in Scheme programs:

- An identifier can be used as a variable or as a syntactic keyword.
- When an identifier appears as a literal or within a literal, it denotes a symbol.

Scheme accepts most of the identifiers that other programming languages allow. MIT/GNU Scheme allows all of the identifiers that standard Scheme does, plus many more.

MIT/GNU Scheme defines a potential identifier to be a sequence of non-delimiter characters that does not begin with either of the characters '`#`' or '`,`'. Any such sequence of characters that is not a syntactically valid number (see Chapter 4 [Numbers], page 63) is considered to be a valid identifier. Note that, although it is legal for '`#`' and '`,`' to appear in an identifier (other than in the first character position), it is poor programming practice.

Here are some examples of identifiers:

```
lambda              q
list->vector        soup
+                   V17a
<=?                 a34kTMNs
the-word-recursion-has-many-meanings
```

1.3.4 Uppercase and Lowercase

Scheme doesn't distinguish uppercase and lowercase forms of a letter except within character and string constants; in other words, Scheme is *case-insensitive*. For example, '`Foo`' is the same identifier as '`FOO`', and '`#x1AB`' is the same number as '`#X1ab`'. But '`#\a`' and '`#\A`' are different characters.

1.3.5 Naming Conventions

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). By convention, predicates usually have names that end in '`?`'.

A *mutation procedure* is a procedure that alters a data structure. By convention, mutation procedures usually have names that end in '`!`'.

1.3.6 Comments

The beginning of a comment is indicated with a semicolon (`;`). Scheme ignores everything on a line in which a semicolon appears, from the semicolon until the end of the line. The entire comment, including the newline character that terminates it, is treated as whitespace.

An alternative form of comment (sometimes called an *extended comment*) begins with the characters '`#|`' and ends with the characters '`|#`'. This alternative form is an MIT/GNU Scheme extension. As with ordinary comments, all of the characters in an extended comment, including the leading '`#|`' and trailing '`|#`', are treated as whitespace. Comments of this form may extend over multiple lines, and additionally may be nested (unlike the comments of the programming language C, which have a similar syntax).

```
;;; This is a comment about the FACT procedure.  Scheme
;;; ignores all of this comment.  The FACT procedure computes
;;; the factorial of a non-negative integer.

#|
This is an extended comment.
Such comments are useful for commenting out code fragments.
|#

(define fact
  (lambda (n)
    (if (= n 0)                     ;This is another comment:
        1                           ;Base case: return 1
        (* n (fact (- n 1))))))
```

1.3.7 Additional Notations

The following list describes additional notations used in Scheme. See Chapter 4 [Numbers], page 63, for a description of the notations used for numbers.

`+ - .` The plus sign, minus sign, and period are used in numbers, and may also occur in an identifier. A delimited period (not occurring within a number or identifier) is used in the notation for pairs and to indicate a "rest" parameter in a formal parameter list (see Section 2.1 [Lambda Expressions], page 15).

`( )` Parentheses are used for grouping and to notate lists (see Chapter 7 [Lists], page 139).

`"` The double quote delimits strings (see Chapter 6 [Strings], page 117).

`\` The backslash is used in the syntax for character constants (see Chapter 5 [Characters], page 109) and as an escape character within string constants (see Chapter 6 [Strings], page 117).

`;` The semicolon starts a comment.

`'` The single quote indicates literal data; it suppresses evaluation (see Section 2.6 [Quoting], page 24).

`` ` `` The backquote indicates almost-constant data (see Section 2.6 [Quoting], page 24).

`,` The comma is used in conjunction with the backquote (see Section 2.6 [Quoting], page 24).

`,@` A comma followed by an at-sign is used in conjunction with the backquote (see Section 2.6 [Quoting], page 24).

`#` The sharp (or pound) sign has different uses, depending on the character that immediately follows it:

`#t #f` These character sequences denote the boolean constants (see Section 10.1 [Booleans], page 167).

`#\` This character sequence introduces a character constant (see Chapter 5 [Characters], page 109).

`#(` This character sequence introduces a vector constant (see Chapter 8 [Vectors], page 157). A close parenthesis, '`)`', terminates a vector constant.

`#e #i #b #o #d #l #s #x`
These character sequences are used in the notation for numbers (see Chapter 4 [Numbers], page 63).

`#|` This character sequence introduces an extended comment. The comment is terminated by the sequence '`|#`'. This notation is an MIT/GNU Scheme extension.

`#!` This character sequence is used to denote a small set of named constants. Currently there are only two of these, `#!optional` and `#!rest`, both of which are used in the `lambda` special form to mark certain parameters as being "optional" or "rest" parameters. This notation is an MIT/GNU Scheme extension.

`#*` This character sequence introduces a bit string (see Chapter 9 [Bit Strings], page 163). This notation is an MIT/GNU Scheme extension.

`#[` This character sequence is used to denote objects that do not have a readable external representation (see Section 14.10 [Custom Output], page 246). A close bracket, '`]`', terminates the object's notation. This notation is an MIT/GNU Scheme extension.

`#@` This character sequence is a convenient shorthand used to refer to objects by their hash number (see Section 14.10 [Custom Output], page 246). This notation is an MIT/GNU Scheme extension.

`#=`
`##` These character sequences introduce a notation used to show circular structures in printed output, or to denote them in input. The notation works much like that in Common Lisp, and is an MIT/GNU Scheme extension.

1.4 Expressions

A Scheme *expression* is a construct that returns a value. An expression may be a *literal*, a *variable reference*, a *special form*, or a *procedure call*.

1.4.1 Literal Expressions

Literal constants may be written by using an external representation of the data. In general, the external representation must be *quoted* (see Section 2.6 [Quoting], page 24); but some external representations can be used without quotation.

```
"abc"                                   ⇒  "abc"
145932                                  ⇒  145932
#t                                      ⇒  #t
#\a                                     ⇒  #\a
```

The external representation of numeric constants, string constants, character constants, and boolean constants evaluate to the constants themselves. Symbols, pairs, lists, and vectors require quoting.

1.4.2 Variable References

An expression consisting of an identifier (see Section 1.3.3 [Identifiers], page 10) is a *variable reference*; the identifier is the name of the variable being referenced. The value of the variable reference is the value stored in the location to which the variable is bound. An error is signalled if the referenced variable is unbound or unassigned.

```
(define x 28)
x                                       ⇒  28
```

1.4.3 Special Form Syntax

```
(keyword component ...)
```

A parenthesized expression that starts with a *syntactic keyword* is a *special form*. Each special form has its own syntax, which is described later in the manual.

Note that syntactic keywords and variable bindings share the same namespace. A local variable binding may shadow a syntactic keyword, and a local syntactic-keyword definition may shadow a variable binding.

The following list contains all of the syntactic keywords that are defined when MIT/GNU Scheme is initialized:

access	and	begin
case	cond	cons-stream
declare	define	
define-integrable	define-structure	define-syntax
delay	do	er-macro-transformer
fluid-let	if	lambda
let	let*	let*-syntax
let-syntax	letrec	letrec-syntax
local-declare	named-lambda	non-hygienic-macro-transformer
or	quasiquote	quote
rsc-macro-transformer	sc-macro-transformer	set!
syntax-rules	the-environment	

1.4.4 Procedure Call Syntax

```
(operator operand ...)
```

A *procedure call* is written by simply enclosing in parentheses expressions for the procedure to be called (the *operator*) and the arguments to be passed to it (the *operands*). The *operator* and *operand* expressions are evaluated and the resulting procedure is passed the resulting arguments. See Section 2.1 [Lambda Expressions], page 15, for a more complete description of this.

Another name for the procedure call expression is *combination*. This word is more specific in that it always refers to the expression; "procedure call" sometimes refers to the *process* of calling a procedure.

Unlike some other dialects of Lisp, Scheme always evaluates the operator expression and the operand expressions with the same evaluation rules, and the order of evaluation is unspecified.

```
(+ 3 4)                                 ⇒  7
((if #f = *) 3 4)                       ⇒  12
```

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables `+` and `*`. New procedures are created by evaluating `lambda` expressions.

If the *operator* is a syntactic keyword, then the expression is not treated as a procedure call: it is a special form.

2 Special Forms

A special form is an expression that follows special evaluation rules. This chapter describes the basic Scheme special forms.

2.1 Lambda Expressions

`lambda` *formals expr expr ...* [extended standard special form]

A `lambda` expression evaluates to a procedure. The environment in effect when the `lambda` expression is evaluated is remembered as part of the procedure; it is called the *closing environment*. When the procedure is later called with some arguments, the closing environment is extended by binding the variables in the formal parameter list to fresh locations, and the locations are filled with the arguments according to rules about to be given. The new environment created by this process is referred to as the *invocation environment*.

Once the invocation environment has been constructed, the *expr*s in the body of the `lambda` expression are evaluated sequentially in it. This means that the region of the variables bound by the `lambda` expression is all of the *expr*s in the body. The result of evaluating the last *expr* in the body is returned as the result of the procedure call.

Formals, the formal parameter list, is often referred to as a *lambda list*.

The process of matching up formal parameters with arguments is somewhat involved. There are three types of parameters, and the matching treats each in sequence:

Required
: All of the *required* parameters are matched against the arguments first. If there are fewer arguments than required parameters, an error of type `condition-type:wrong-number-of-arguments` is signalled; this error is also signalled if there are more arguments than required parameters and there are no further parameters.

Optional
: Once the required parameters have all been matched, the *optional* parameters are matched against the remaining arguments. If there are fewer arguments than optional parameters, the unmatched parameters are bound to special objects called *default objects*. If there are more arguments than optional parameters, and there are no further parameters, an error of type `condition-type:wrong-number-of-arguments` is signalled.

  The predicate `default-object?`, which is true only of default objects, can be used to determine which optional parameters were supplied, and which were defaulted.

Rest
: Finally, if there is a *rest* parameter (there can only be one), any remaining arguments are made into a list, and the list is bound to the rest parameter. (If there are no remaining arguments, the rest parameter is bound to the empty list.)

  In Scheme, unlike some other Lisp implementations, the list to which a rest parameter is bound is always freshly allocated. It has infinite extent and may be modified without affecting the procedure's caller.

Specially recognized keywords divide the *formals* parameters into these three classes. The keywords used here are '`#!optional`', '`.`', and '`#!rest`'. Note that only '`.`' is defined by standard Scheme — the other keywords are MIT/GNU Scheme extensions. '`#!rest`' has the same meaning as '`.`' in *formals*.

The use of these keywords is best explained by means of examples. The following are typical lambda lists, followed by descriptions of which parameters are required, optional, and rest. We will use '`#!rest`' in these examples, but anywhere it appears '`.`' could be used instead.

`(a b c)`
: `a`, `b`, and `c` are all required. The procedure must be passed exactly three arguments.

`(a b #!optional c)`
: `a` and `b` are required, `c` is optional. The procedure may be passed either two or three arguments.

`(#!optional a b c)`
: `a`, `b`, and `c` are all optional. The procedure may be passed any number of arguments between zero and three, inclusive.

`a`
`(#!rest a)`
: These two examples are equivalent. `a` is a rest parameter. The procedure may be passed any number of arguments. Note: this is the only case in which '`.`' cannot be used in place of '`#!rest`'.

`(a b #!optional c d #!rest e)`
: `a` and `b` are required, `c` and `d` are optional, and `e` is rest. The procedure may be passed two or more arguments.

Some examples of `lambda` expressions:

```
(lambda (x) (+ x x))            ⇒  #[compound-procedure 53]

((lambda (x) (+ x x)) 4)                ⇒  8

(define reverse-subtract
  (lambda (x y)
    (- y x)))
(reverse-subtract 7 10)                 ⇒  3

(define foo
  (let ((x 4))
    (lambda (y) (+ x y))))
(foo 6)                                 ⇒  10
```

`named-lambda` *formals expression expression . . .* [special form]

The `named-lambda` special form is similar to `lambda`, except that the first "required parameter" in *formals* is not a parameter but the *name* of the resulting procedure; thus *formals* must have at least one required parameter. This name has no semantic meaning, but is included in the external representation of the procedure, making

it useful for debugging. In MIT/GNU Scheme, `lambda` is implemented as `named-lambda`, with a special name that means "unnamed".

```
(named-lambda (f x) (+ x x))    ⇒  #[compound-procedure 53 f]
((named-lambda (f x) (+ x x)) 4)        ⇒  8
```

2.2 Lexical Binding

The binding constructs `let`, `let*`, `letrec`, `letrec*`, `let-values`, and `let*-values` give Scheme block structure, like Algol 60. The syntax of the first four constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially; while in `letrec` and `letrec*` expressions, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. The `let-values` and `let*-values` constructs are analogous to `let` and `let*` respectively, but are designed to handle multiple-valued expressions, binding different identifiers to the returned values.

`let` ((*variable init*) ...) *expr expr ...* [extended standard special form]
The *init*s are evaluated in the current environment (in some unspecified order), the *variable*s are bound to fresh locations holding the results, the *expr*s are evaluated sequentially in the extended environment, and the value of the last *expr* is returned. Each binding of a *variable* has the *expr*s as its region.

MIT/GNU Scheme allows any of the *init*s to be omitted, in which case the corresponding *variable*s are unassigned.

Note that the following are equivalent:

```
(let ((variable init) ...) expr expr ...)
((lambda (variable ...) expr expr ...) init ...)
```

Some examples:

```
(let ((x 2) (y 3))
  (* x y))                              ⇒  6

(let ((x 2) (y 3))
  (let ((foo (lambda (z) (+ x y z)))
        (x 7))
    (foo 4)))                           ⇒  9
```

See Section 2.9 [Iteration], page 29, for information on "named `let`".

`let*` ((*variable init*) ...) *expr expr ...* [extended standard special form]
`let*` is similar to `let`, but the bindings are performed sequentially from left to right, and the region of a binding is that part of the `let*` expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

Note that the following are equivalent:

```
(let* ((variable1 init1)
       (variable2 init2)
       ...
       (variableN initN))
   expr
   expr ...)

(let ((variable1 init1))
  (let ((variable2 init2))
    ...
      (let ((variableN initN))
        expr
        expr ...)
    ...))
```

An example:

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                          ⇒  70
```

`letrec` ((*variable init*) ...) *expr expr* ... [extended standard special form]

The *variable*s are bound to fresh locations holding unassigned values, the *init*s are evaluated in the extended environment (in some unspecified order), each *variable* is assigned to the result of the corresponding *init*, the *expr*s are evaluated sequentially in the extended environment, and the value of the last *expr* is returned. Each binding of a *variable* has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

MIT/GNU Scheme allows any of the *init*s to be omitted, in which case the corresponding *variable*s are unassigned.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))                          ⇒  #t
```

One restriction on `letrec` is very important: it shall be possible to evaluated each *init* without assigning or referring to the value of any *variable*. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name. In the most common uses of `letrec`, all the *init*s are `lambda` or `delay` expressions and the restriction is satisfied automatically.

`letrec*` ((*variable init*) ...) *expr expr ...* [extended standard special form]

The *variable*s are bound to fresh locations, each *variable* is assigned in left-to-right order to the result of evaluating the corresponding *init* (interleaving evaluations and assignments), the *expr*s are evaluated in the resulting environment, and the values of the last *expr* are returned. Despite the left-to-right evaluation and assignment order, each binding of a *variable* has the entire `letrec*` expression as its region, making it possible to define mutually recursive procedures.

If it is not possible to evaluate each *init* without assigning or referring to the value of the corresponding *variable* or the *variable* of any of the bindings that follow it in *bindings*, it is an error. Another restriction is that it is an error to invoke the continuation of an *init* more than once.

```
;; Returns the arithmetic, geometric, and
;; harmonic means of a nested list of numbers
(define (means ton)
  (letrec*
     ((mean
        (lambda (f g)
          (f (/ (sum g ton) n))))
      (sum
        (lambda (g ton)
          (if (null? ton)
            (+)
            (if (number? ton)
                (g ton)
                (+ (sum g (car ton))
                   (sum g (cdr ton)))))))
      (n (sum (lambda (x) 1) ton)))
    (values (mean values values)
            (mean exp log)
            (mean / /))))
```

Evaluating `(means '(3 (1 4)))` returns three values: `8/3`, `2.28942848510666` (approximately), and `36/19`.

`let-values` ((*formals init*) ...) *expr expr ...* [standard special form]

The *init*s are evaluated in the current environment (in some unspecified order) as if by invoking `call-with-values`, and the variables occurring in the *formals* are bound to fresh locations holding the values returned by the *init*s, where the *formals* are matched to the return values in the same way that the *formals* in a `lambda` expression are matched to the arguments in a procedure call. Then, the *expr*s are evaluated in the extended environment, and the values of the last *expr* are returned. Each binding of a *variable* has the *expr*s as its region.

It is an error if the *formals* do not match the number of values returned by the corresponding *init*.

```
(let-values (((root rem) (exact-integer-sqrt 32)))
  (* root rem))         ⇒  35
```

`let*-values` ((*formals init*) ...) *expr expr ...* [standard special form]
The `let*-values` construct is similar to `let-values`, but the *init*s are evaluated and bindings created sequentially from left to right, with the region of the bindings of each *formals* including the *init*s to its right as well as *body*. Thus the second *init* is evaluated in an environment in which the first set of bindings is visible and initialized, and so on.

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y)))    ⇒  (x y x y)
```

2.3 Dynamic Binding

`parameterize` ((*parameter value*) ...) *expr expr ...* [standard special form]
Note that both *parameter* and *value* are expressions. It is an error if the value of any *parameter* expression is not a parameter object.

A `parameterize` expression is used to change the values of specified parameter objects during the evaluation of the body *expression*s.

The *parameter* and *value* expressions are evaluated in an unspecified order. The body is evaluated in a dynamic environment in which each *parameter* is bound to the converted *value*—the result of passing *value* to the conversion procedure specified when the *parameter* was created. Then the previous value of *parameter* is restored without passing it to the conversion procedure. The value of the parameterize expression is the value of the last body *expr*.

The `parameterize` special form is standardized by SRFI 39 and by R7RS.

Parameter objects can be used to specify configurable settings for a computation without the need to pass the value to every procedure in the call chain explicitly.

```
(define radix
  (make-parameter
   10
   (lambda (x)
     (if (and (exact-integer?  x) (<= 2 x 16))
         x
         (error "invalid radix")))))

(define (f n) (number->string n (radix)))

(f 12)                                  ⇒ "12"
(parameterize ((radix 2))
  (f 12))                               ⇒ "1100"
(f 12)                                  ⇒ "12"
(radix 16)                              error  Wrong number of arguments
(parameterize ((radix 0))
  (f 12))                               error  invalid radix
```

A *dynamic binding* changes the value of a parameter (see Section 10.3 [Parameters], page 171) object temporarily, for a *dynamic extent*. The set of all dynamic bindings at a given time is called the *dynamic environment*. The new values are only accessible to the thread that constructed the dynamic environment, and any threads created within that environment.

The *extent* of a dynamic binding is defined to be the time period during which calling the parameter returns the new value. Normally this time period begins when the body is entered and ends when it is exited, a contiguous time period. However Scheme has first-class continuations by which it is possible to leave the body and reenter it many times. In this situation, the extent is non-contiguous.

When the body is exited by invoking a continuation, the current dynamic environment is unwound until it can be re-wound to the environment captured by the continuation. When the continuation returns, the process is reversed, restoring the original dynamic environment.

The following example shows the interaction between dynamic binding and continuations. Side effects to the binding that occur both inside and outside of the body are preserved, even if continuations are used to jump in and out of the body repeatedly.

```
(define (complicated-dynamic-parameter)
  (let ((variable (make-settable-parameter 1))
        (inside-continuation))
    (write-line (variable))
    (call-with-current-continuation
     (lambda (outside-continuation)
       (parameterize ((variable 2))
         (write-line (variable))
         (variable 3)
         (call-with-current-continuation
          (lambda (k)
            (set! inside-continuation k)
            (outside-continuation #t)))
         (write-line (variable))
         (set! inside-continuation #f))))
    (write-line (variable))
    (if inside-continuation
        (begin
          (variable 4)
          (inside-continuation #f)))))
```

Evaluating '`(complicated-dynamic-binding)`' writes the following on the console:

```
1
2
1
3
4
```

Commentary: the first two values written are the initial binding of `variable` and its new binding inside `parameterize`'s body. Immediately after they are written, the binding visible

in the body is set to '3', and `outside-continuation` is invoked, exiting the body. At this point, '1' is written, demonstrating that the original binding of `variable` is still visible outside the body. Then we set `variable` to '4' and reenter the body by invoking `inside-continuation`. At this point, '3' is written, indicating that the binding modified in the body is still the binding visible in the body. Finally, we exit the body normally, and write '4', demonstrating that the binding modified outside of the body was also preserved.

2.3.1 Fluid-Let

The `fluid-let` special form can change the value of *any* variable for a dynamic extent, but it is difficult to implement in a multi-processing (SMP) world. It and the cell object type (see [Cells], page 172) are now **deprecated**. They are still available and functional in a uni-processing (non-SMP) world, but will signal an error when used in an SMP world. The `parameterize` special form (see [parameterize], page 20) should be used instead.

`fluid-let` ((*variable init*) ...) *expression expression ...* [special form]

The *init*s are evaluated in the current environment (in some unspecified order), the current values of the *variable*s are saved, the results are assigned to the *variable*s, the *expression*s are evaluated sequentially in the current environment, the *variable*s are restored to their original values, and the value of the last *expression* is returned.

The syntax of this special form is similar to that of `let`, but `fluid-let` temporarily rebinds existing variables. Unlike `let`, `fluid-let` creates no new bindings; instead it *assigns* the value of each *init* to the binding (determined by the rules of lexical scoping) of its corresponding *variable*.

MIT/GNU Scheme allows any of the *init*s to be omitted, in which case the corresponding *variable*s are temporarily unassigned.

An error of type `condition-type:unbound-variable` is signalled if any of the *variable*s are unbound. However, because `fluid-let` operates by means of side effects, it is valid for any *variable* to be unassigned when the form is entered.

2.4 Definitions

`define` *variable* [*expression*] [extended standard special form]
`define` *formals expression expression ...* [standard special form]

Definitions are valid in some but not all contexts where expressions are allowed. Definitions may only occur at the top level of a program and at the beginning of a lambda body (that is, the body of a `lambda`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `parameterize`, or "procedure `define`" expression). A definition that occurs at the top level of a program is called a *top-level definition*, and a definition that occurs at the beginning of a body is called an *internal definition*.

In the second form of `define` (called "*procedure* `define`"), the component *formals* is identical to the component of the same name in a `named-lambda` expression. In fact, these two expressions are equivalent:

```
(define (name1 name2 ...)
  expression
  expression ...)
```

```
(define name1
  (named-lambda (name1 name2 ...)
    expression
    expression ...))
```

2.4.1 Top-Level Definitions

A top-level definition,

```
(define variable expression)
```

has essentially the same effect as this assignment expression, if *variable* is bound:

```
(set! variable expression)
```

If *variable* is not bound, however, `define` binds *variable* to a new location in the current environment before performing the assignment (it is an error to perform a `set!` on an unbound variable). If you omit *expression*, the variable becomes unassigned; an attempt to reference such a variable is an error.

```
(define add3
   (lambda (x) (+ x 3)))                ⇒  unspecified
(add3 3)                                ⇒  6

(define first car)                      ⇒  unspecified
(first '(1 2))                          ⇒  1

(define bar)                            ⇒  unspecified
bar                                     error  Unassigned variable
```

2.4.2 Internal Definitions

An *internal definition* is a definition that occurs at the beginning of a *body* (that is, the body of a `lambda`, `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `parameterize`, or "procedure `define`" expression), rather than at the top level of a program. The variable defined by an internal definition is local to the *body*. That is, *variable* is bound rather than assigned, and the region of the binding is the entire *body*. For example,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))                        ⇒  45
```

A *body* containing internal definitions can always be converted into a completely equivalent `letrec*` expression. For example, the `let` expression in the above example is equivalent to

```
(let ((x 5))
  (letrec* ((foo (lambda (y) (bar x y)))
            (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

2.5 Assignments

`set!` *variable* [*expression*] [extended standard special form]

If *expression* is specified, evaluates *expression* and stores the resulting value in the location to which *variable* is bound. If *expression* is omitted, *variable* is altered to be unassigned; a subsequent reference to such a *variable* is an error. In either case, the value of the `set!` expression is unspecified.

Variable must be bound either in some region enclosing the `set!` expression, or at the top level. However, *variable* is permitted to be unassigned when the `set!` form is entered.

```
(define x 2)                    ⇒  unspecified
(+ x 1)                         ⇒  3
(set! x 4)                      ⇒  unspecified
(+ x 1)                         ⇒  5
```

Variable may be an `access` expression (see Chapter 13 [Environments], page 223). This allows you to assign variables in an arbitrary environment. For example,

```
(define x (let ((y 0)) (the-environment)))
(define y 'a)
y                               ⇒  a
(access y x)                    ⇒  0
(set! (access y x) 1)           ⇒  unspecified
y                               ⇒  a
(access y x)                    ⇒  1
```

2.6 Quoting

This section describes the expressions that are used to modify or prevent the evaluation of objects.

`quote` *datum* [standard special form]

`(quote` *datum*`)` evaluates to *datum*. *Datum* may be any external representation of a Scheme object (see Section 1.2.6 [External Representations], page 8). Use `quote` to include literal constants in Scheme code.

```
(quote a)                       ⇒  a
(quote #(a b c))                ⇒  #(a b c)
(quote (+ 1 2))                 ⇒  (+ 1 2)
```

`(quote` *datum*`)` may be abbreviated as `'`*datum*. The two notations are equivalent in all respects.

```
'a                              ⇒  a
'#(a b c)                       ⇒  #(a b c)
'(+ 1 2)                        ⇒  (+ 1 2)
'(quote a)                      ⇒  (quote a)
''a                             ⇒  (quote a)
```

Numeric constants, string constants, character constants, and boolean constants evaluate to themselves, so they don't need to be quoted.

```
'"abc"                                  ⇒  "abc"
"abc"                                   ⇒  "abc"
'145932                                 ⇒  145932
145932                                  ⇒  145932
'#t                                     ⇒  #t
#t                                      ⇒  #t
'#\a                                    ⇒  #\a
#\a                                     ⇒  #\a
```

`quasiquote` *template* [standard special form]

"Backquote" or "quasiquote" expressions are useful for constructing a list or vector structure when most but not all of the desired structure is known in advance. If no commas appear within the *template*, the result of evaluating `` `template `` is equivalent (in the sense of `equal?`) to the result of evaluating `'template`. If a comma appears within the *template*, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression shall evaluate to a list; the opening and closing parentheses of the list are then "stripped away" and the elements of the list are inserted in place of the comma at-sign expression sequence.

```
`(list ,(+ 1 2) 4)                      ⇒  (list 3 4)

(let ((name 'a)) `(list ,name ',name))  ⇒  (list a 'a)

`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)   ⇒  (a 3 4 5 6 b)

`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
                                        ⇒  ((foo 7) . cons)

`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
                                        ⇒  #(10 5 2 4 3 8)

`,(+ 2 3)                               ⇒  5
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
     ⇒  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)

(let ((name1 'x)
      (name2 'y))
   `(a `(b ,,name1 ,',name2 d) e))
     ⇒  (a `(b ,x ,'y d) e)
```

The notations `` `template `` and (quasiquote *template*) are identical in all respects. `,expression` is identical to (unquote *expression*) and `,@expression` is identical to (unquote-splicing *expression*).

```
(quasiquote (list (unquote (+ 1 2)) 4))
     ⇒  (list 3 4)

'(quasiquote (list (unquote (+ 1 2)) 4))
     ⇒  `(list ,(+ 1 2) 4)
     i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

Unpredictable behavior can result if any of the symbols `quasiquote`, `unquote`, or `unquote-splicing` appear in a *template* in ways otherwise than as described above.

2.7 Conditionals

The behavior of the *conditional expressions* is determined by whether objects are true or false. The conditional expressions count only `#f` as false. They count everything else, including `#t`, pairs, symbols, numbers, strings, vectors, and procedures as true (but see Section 1.2.5 [True and False], page 8).

In the descriptions that follow, we say that an object has "a true value" or "is true" when the conditional expressions treat it as true, and we say that an object has "a false value" or "is false" when the conditional expressions treat it as false.

`if` *predicate consequent* [*alternative*] [standard special form]

Predicate, *consequent*, and *alternative* are expressions. An `if` expression is evaluated as follows: first, *predicate* is evaluated. If it yields a true value, then *consequent* is evaluated and its value is returned. Otherwise *alternative* is evaluated and its value is returned. If *predicate* yields a false value and no *alternative* is specified, then the result of the expression is unspecified.

An `if` expression evaluates either *consequent* or *alternative*, never both. Programs should not depend on the value of an `if` expression that has no *alternative*.

```
(if (> 3 2) 'yes 'no)                    ⇒  yes
(if (> 2 3) 'yes 'no)                    ⇒  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                             ⇒  1
```

`cond` *clause clause* ... [standard special form]

Each *clause* has this form:

```
(predicate expression ...)
```

where *predicate* is any expression. The last *clause* may be an `else` *clause*, which has the form:

```
(else expression expression ...)
```

A `cond` expression does the following:

1. Evaluates the *predicate* expressions of successive *clause*s in order, until one of the *predicate*s evaluates to a true value.

2. When a *predicate* evaluates to a true value, `cond` evaluates the *expression*s in the associated *clause* in left to right order, and returns the result of evaluating the last *expression* in the *clause* as the result of the entire `cond` expression.

   If the selected *clause* contains only the *predicate* and no *expression*s, `cond` returns the value of the *predicate* as the result.

3. If all *predicate*s evaluate to false values, and there is no `else` clause, the result of the conditional expression is unspecified; if there is an `else` clause, `cond` evaluates its *expression*s (left to right) and returns the value of the last one.

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))                  ⇒  greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))                    ⇒  equal
```

Normally, programs should not depend on the value of a `cond` expression that has no `else` clause. However, some Scheme programmers prefer to write `cond` expressions in which at least one of the *predicate*s is always true. In this style, the final *clause* is equivalent to an `else` clause.

Scheme supports an alternative *clause* syntax:

```
(predicate => recipient)
```

where *recipient* is an expression. If *predicate* evaluates to a true value, then *recipient* is evaluated. Its value must be a procedure of one argument; this procedure is then invoked on the value of the *predicate*.

```
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))                        ⇒  2
```

`case` *key clause clause . . .* [standard special form]

Key may be any expression. Each *clause* has this form:

```
((object ...) expression expression ...)
```

No *object* is evaluated, and all the *object*s must be distinct. The last *clause* may be an `else` *clause*, which has the form:

```
(else expression expression ...)
```

A `case` expression does the following:

1. Evaluates *key* and compares the result with each *object*.
2. If the result of evaluating *key* is equivalent (in the sense of `eqv?`; see Chapter 3 [Equivalence Predicates], page 57) to an *object*, `case` evaluates the *expression*s in the corresponding *clause* from left to right and returns the result of evaluating the last *expression* in the *clause* as the result of the `case` expression.
3. If the result of evaluating *key* is different from every *object*, and if there's an `else` clause, `case` evaluates its *expression*s and returns the result of the last one as the result of the `case` expression. If there's no `else` clause, `case` returns an unspecified result. Programs should not depend on the value of a `case` expression that has no `else` clause.

For example,

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

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

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

`and` *expression ...* [standard special form]

The *expression*s are evaluated from left to right, and the value of the first *expression* that evaluates to a false value is returned. Any remaining *expression*s are not evaluated. If all the *expression*s evaluate to true values, the value of the last *expression* is returned. If there are no *expression*s then `#t` is returned.

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

`or` *expression ...* [standard special form]

The *expression*s are evaluated from left to right, and the value of the first *expression* that evaluates to a true value is returned. Any remaining *expression*s are not evaluated. If all *expression*s evaluate to false values, the value of the last *expression* is returned. If there are no *expression*s then `#f` is returned.

```
(or (= 2 2) (> 2 1))                   ⇒  #t
(or (= 2 2) (< 2 1))                   ⇒  #t
(or #f #f #f)                          ⇒  #f
(or (memq 'b '(a b c)) (/ 3 0))        ⇒  (b c)
```

2.8 Sequencing

The `begin` special form is used to evaluate expressions in a particular order.

`begin` *expression expression ...* [standard special form]

The *expression*s are evaluated sequentially from left to right, and the value of the last *expression* is returned. This expression type is used to sequence side effects such as input and output.

```
(define x 0)
(begin (set! x 5)
       (+ x 1))                 ⇒  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))
                                ⊣  4 plus 1 equals 5
                                ⇒  unspecified
```

Often the use of `begin` is unnecessary, because many special forms already support sequences of expressions (that is, they have an implicit `begin`). Some of these special forms are:

```
case
cond
define          ;"procedure define" only
do
lambda
let
let*
letrec
letrec*
let-values
let*-values
named-lambda
parameterize
```

2.9 Iteration

The *iteration expressions* are: "named `let`" and `do`. They are also binding expressions, but are more commonly referred to as iteration expressions. Because Scheme is properly tail-recursive, you don't need to use these special forms to express iteration; you can simply use appropriately written "recursive" procedure calls.

`let` *name* ((*variable init*) ...) *expr expr* ... [extended standard special form]

MIT/GNU Scheme permits a variant on the syntax of `let` called "named `let`" which provides a more general looping construct than `do`, and may also be used to express recursions.

Named `let` has the same syntax and semantics as ordinary `let` except that *name* is bound within the *expr*s to a procedure whose formal arguments are the *variable*s and whose body is the *expr*s. Thus the execution of the *expr*s may be repeated by invoking the procedure named by *name*.

MIT/GNU Scheme allows any of the *init*s to be omitted, in which case the corresponding *variable*s are unassigned.

Note: the following expressions are equivalent:

```
(let name ((variable init) ...)
  expr
  expr ...)

((letrec ((name
           (named-lambda (name variable ...)
             expr
             expr ...)))
   name)
 init ...)
```

Here is an example:

```
(let loop
     ((numbers '(3 -2 1 6 -5))
      (nonneg '())
      (neg '()))
  (cond ((null? numbers)
         (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        (else
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))

    ⇒  ((6 1 3) (-5 -2))
```

`do` ((*variable init step*) ...) (*test expression* ...) *command* ... [extended standard special form]

`do` is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits with a specified result value.

`do` expressions are evaluated as follows: The *init* expressions are evaluated (in some unspecified order), the *variable*s are bound to fresh locations, the results of the *init* expressions are stored in the bindings of the *variable*s, and then the iteration phase begins.

Each iteration begins by evaluating *test*; if the result is false, then the *command* expressions are evaluated in order for effect, the *step* expressions are evaluated in some unspecified order, the *variable*s are bound to fresh locations, the results of the *step*s are stored in the bindings of the *variable*s, and the next iteration begins.

If *test* evaluates to a true value, then the *expression*s are evaluated from left to right and the value of the last *expression* is returned as the value of the `do` expression. If no *expression*s are present, then the value of the `do` expression is unspecified in standard Scheme; in MIT/GNU Scheme, the value of *test* is returned.

The region of the binding of a *variable* consists of the entire `do` expression except for the *init*s. It is an error for a *variable* to appear more than once in the list of `do` variables.

A *step* may be omitted, in which case the effect is the same as if `(`*`variable init variable`*`)` had been written instead of `(`*`variable init`*`)`.

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))              ⇒  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))               ⇒  25
```

2.10 Structure Definitions

This section provides examples and describes the options and syntax of `define-structure`, an MIT/GNU Scheme macro that is very similar to `defstruct` in Common Lisp. The differences between them are summarized at the end of this section. For more information, see Steele's Common Lisp book.

`define-structure` (*name structure-option ...*) *slot-description ...* [special form]

Each *slot-description* takes one of the following forms:

```
slot-name
(slot-name default-init [slot-option value]*)
```

The fields *name* and *slot-name* must both be symbols. The field *default-init* is an expression for the initial value of the slot. It is evaluated each time a new instance is constructed. If it is not specified, the initial content of the slot is undefined. Default values are only useful with a BOA constructor with argument list or a keyword constructor (see below).

Evaluation of a `define-structure` expression defines a structure descriptor and a set of procedures to manipulate instances of the structure. These instances are represented as records by default (see Section 10.4 [Records], page 172) but may alternately be lists or vectors. The accessors and modifiers are marked with compiler declarations so that calls to them are automatically transformed into appropriate references. Often, no options are required, so a simple call to `define-structure` looks like:

```
(define-structure foo a b c)
```

This defines a type descriptor `rtd:foo`, a constructor `make-foo`, a predicate `foo?`, accessors `foo-a`, `foo-b`, and `foo-c`, and modifiers `set-foo-a!`, `set-foo-b!`, and `set-foo-c!`.

In general, if no options are specified, `define-structure` defines the following (using the simple call above as an example):

type descriptor
: The name of the type descriptor is `"rtd:"` followed by the name of the structure, e.g. '`rtd:foo`'. The type descriptor satisfies the predicate `record-type?`.

constructor
: The name of the constructor is `"make-"` followed by the name of the structure, e.g. '`make-foo`'. The number of arguments accepted by the constructor is the same as the number of slots; the arguments are the initial values for the slots, and the order of the arguments matches the order of the slot definitions.

predicate
: The name of the predicate is the name of the structure followed by `"?"`, e.g. '`foo?`'. The predicate is a procedure of one argument, which returns `#t` if its argument is a record of the type defined by this structure definition, and `#f` otherwise.

accessors
: For each slot, an accessor is defined. The name of the accessor is formed by appending the name of the structure, a hyphen, and the name of the slot, e.g. '`foo-a`'. The accessor is a procedure of one argument, which must be a record of the type defined by this structure definition. The accessor extracts the contents of the corresponding slot in that record and returns it.

modifiers
: For each slot, a modifier is defined. The name of the modifier is formed by appending `"set-"`, the name of the accessor, and `"!"`, e.g. '`set-foo-a!`'. The modifier is a procedure of two arguments, the first of which must be a record of the type defined by this structure definition, and the second of which may be any object. The modifier modifies the contents of the corresponding slot in that record to be that object, and returns an unspecified value.

When options are not supplied, (*name*) may be abbreviated to *name*. This convention holds equally for *structure-options* and *slot-options*. Hence, these are equivalent:

```
(define-structure foo a b c)
(define-structure (foo) (a) b (c))
```

as are

```
(define-structure (foo keyword-constructor) a b c)
(define-structure (foo (keyword-constructor)) a b c)
```

When specified as option values, `false` and `nil` are equivalent to `#f`, and `true` and `t` are equivalent to `#t`.

Possible *slot-options* are:

`read-only` *value* [slot option]
: When given a *value* other than `#f`, this specifies that no modifier should be created for the slot.

`type` *type-descriptor* [slot option]
: This is accepted but not presently used.

Possible *structure-options* are:

`predicate` [*name*] [structure option]
: This option controls the definition of a predicate procedure for the structure. If *name* is not given, the predicate is defined with the default name (see above). If *name* is `#f`, the predicate is not defined at all. Otherwise, *name* must be a symbol, and the predicate is defined with that symbol as its name.

`copier` [*name*] [structure option]
: This option controls the definition of a procedure to copy instances of the structure. This is a procedure of one argument, a structure instance, that makes a newly allocated copy of the structure and returns it. If *name* is not given, the copier is defined, and the name of the copier is `"copy-"` followed by the structure name (e.g. '`copy-foo`'). If *name* is `#f`, the copier is not defined. Otherwise, *name* must be a symbol, and the copier is defined with that symbol as its name.

`print-procedure` *expression* [structure option]
: Evaluating *expression* must yield a procedure of two arguments, which is used to print instances of the structure. The procedure is a *print method* (see Section 14.10 [Custom Output], page 246).

`constructor` [*name* [*argument-list*]] [structure option]
: This option controls the definition of constructor procedures. These constructor procedures are called "BOA constructors", for "By Order of Arguments", because the arguments to the constructor specify the initial contents of the structure's slots by the order in which they are given. This is as opposed to "keyword constructors", which specify the initial contents using keywords, and in which the order of arguments is irrelevant.

  If *name* is not given, a constructor is defined with the default name and arguments (see above). If *name* is `#f`, no constructor is defined; *argument-list* may not be specified in this case. Otherwise, *name* must be a symbol, and a constructor is defined with that symbol as its name. If *name* is a symbol, *argument-list* is optionally allowed; if it is omitted, the constructor accepts one argument for each slot in the structure definition, in the same order in which the slots appear in the definition. Otherwise, *argument-list* must be a lambda list (see Section 2.1 [Lambda Expressions], page 15), and each of the parameters of the lambda list must be the name of a slot in the structure. The arguments accepted by the constructor are defined by this lambda list. Any slot that is not specified by the lambda list is initialized to the *default-init* as specified above; likewise for any slot specified as an optional parameter when the corresponding argument is not supplied.

  If the `constructor` option is specified, the default constructor is not defined. Additionally, the `constructor` option may be specified multiple times to define multiple constructors with different names and argument lists.

```
(define-structure (foo
                   (constructor make-foo (#!optional a b)))
  (a 6 read-only #t)
  (b 9))
```

`keyword-constructor` [*name*] [structure option]

This option controls the definition of keyword constructor procedures. A *keyword constructor* is a procedure that accepts arguments that are alternating slot names and values. If *name* is omitted, a keyword constructor is defined, and the name of the constructor is `"make-"` followed by the name of the structure (e.g. '`make-foo`'). Otherwise, *name* must be a symbol, and a keyword constructor is defined with this symbol as its name.

If the `keyword-constructor` option is specified, the default constructor is not defined. Additionally, the `keyword-constructor` option may be specified multiple times to define multiple keyword constructors; this is usually not done since such constructors would all be equivalent.

```
(define-structure (foo (keyword-constructor make-bar)) a b)
(foo-a (make-bar 'b 20 'a 19))          ⇒ 19
```

`type-descriptor` *name* [structure option]

This option cannot be used with the `type` or `named` options.

By default, structures are implemented as records. The name of the structure is defined to hold the type descriptor of the record defined by the structure. The `type-descriptor` option specifies a different name to hold the type descriptor.

```
(define-structure foo a b)
foo              ⇒ #[record-type 18]

(define-structure (bar (type-descriptor <bar>)) a b)
bar              error  Unbound variable: bar
<bar>          ⇒ #[record-type 19]
```

`conc-name` [*name*] [structure option]

By default, the prefix for naming accessors and modifiers is the name of the structure followed by a hyphen. The `conc-name` option can be used to specify an alternative. If *name* is not given, the prefix is the name of the structure followed by a hyphen (the default). If *name* is `#f`, the slot names are used directly, without prefix. Otherwise, *name* must a symbol, and that symbol is used as the prefix.

```
(define-structure (foo (conc-name moby/)) a b)
```

defines accessors `moby/a` and `moby/b`, and modifiers `set-moby/a!` and `set-moby/b!`.

```
(define-structure (foo (conc-name #f)) a b)
```

defines accessors `a` and `b`, and modifiers `set-a!` and `set-b!`.

`type` *representation-type* [structure option]

This option cannot be used with the `type-descriptor` option.

By default, structures are implemented as records. The `type` option overrides this default, allowing the programmer to specify that the structure be implemented using another data type. The option value *representation-type* specifies the alternate data type; it is allowed to be one of the symbols `vector` or `list`, and the data type used is the one corresponding to the symbol.

If this option is given, and the `named` option is not specified, the representation will not be tagged, and neither a predicate nor a type descriptor will be defined; also, the `print-procedure` option may not be given.

```
(define-structure (foo (type list)) a b)
(make-foo 1 2)                          ⇒ (1 2)
```

`named` [*expression*] [structure option]

This is valid only in conjunction with the `type` option and specifies that the structure instances be tagged to make them identifiable as instances of this structure type. This option cannot be used with the `type-descriptor` option.

In the usual case, where *expression* is not given, the `named` option causes a type descriptor and predicate to be defined for the structure (recall that the `type` option without `named` suppresses their definition), and also defines a default print method for the structure instances (which can be overridden by the `print-procedure` option). If the default print method is not wanted then the `print-procedure` option should be specified as `#f`. This causes the structure to be printed in its native representation, as a list or vector, which includes the type descriptor. The type descriptor is a unique object, *not* a record type, that describes the structure instances and is additionally stored in the structure instances to identify them: if the representation type is `vector`, the type descriptor is stored in the zero-th slot of the vector, and if the representation type is `list`, it is stored as the first element of the list.

```
(define-structure (foo (type vector) named) a b c)
(vector-ref (make-foo 1 2 3) 0) ⇒ #[structure-type 52]
```

If *expression* is specified, it is an expression that is evaluated to yield a tag object. The *expression* is evaluated once when the structure definition is evaluated (to specify the print method), and again whenever a predicate or constructor is called. Because of this, *expression* is normally a variable reference or a constant. The value yielded by *expression* may be any object at all. That object is stored in the structure instances in the same place that the type descriptor is normally stored, as described above. If *expression* is specified, no type descriptor is defined, only a predicate.

```
(define-structure (foo (type vector) (named 'foo)) a b c)
(vector-ref (make-foo 1 2 3) 0) ⇒ foo
```

`safe-accessors` [*boolean*] [structure option]

This option allows the programmer to have some control over the safety of the slot accessors (and modifiers) generated by `define-structure`. If `safe-accessors` is not specified, or if *boolean* is `#f`, then the accessors are optimized for speed at the expense of safety; when compiled, the accessors will turn into very fast inline sequences, usually one to three machine instructions in length. However, if `safe-accessors` is specified and *boolean* is either omitted or `#t`, then the accessors are optimized for safety, will check the type and structure of their argument, and will be close-coded.

```
(define-structure (foo safe-accessors) a b c)
```

`initial-offset` *offset* [structure option]

This is valid only in conjunction with the `type` option. *Offset* must be an exact non-negative integer and specifies the number of slots to leave open at the beginning of

the structure instance before the specified slots are allocated. Specifying an *offset* of zero is equivalent to omitting the `initial-offset` option.

If the `named` option is specified, the structure tag appears in the first slot, followed by the "offset" slots, and then the regular slots. Otherwise, the "offset" slots come first, followed by the regular slots.

```
(define-structure (foo (type vector) (initial-offset 3))
  a b c)
(make-foo 1 2 3)                ⇒ #(() () () 1 2 3)
```

The essential differences between MIT/GNU Scheme's `define-structure` and Common Lisp's `defstruct` are:

- The default constructor procedure takes positional arguments, in the same order as specified in the definition of the structure. A keyword constructor may be specified by giving the option `keyword-constructor`.
- BOA constructors are described using Scheme lambda lists. Since there is nothing corresponding to `&aux` in Scheme lambda lists, this functionality is not implemented.
- By default, no `copier` procedure is defined.
- The side-effect procedure corresponding to the accessor `foo` is given the name `set-foo!`.
- Keywords are ordinary symbols – use `foo` instead of `:foo`.
- The option values `false`, `nil`, `true`, and `t` are treated as if the appropriate boolean constant had been specified instead.
- The `print-function` option is named `print-procedure`. Its argument is a procedure of two arguments (the structure instance and a textual output port) rather than three as in Common Lisp.
- By default, named structures are tagged with a unique object of some kind. In Common Lisp, the structures are tagged with symbols. This depends on the Common Lisp package system to help generate unique tags; MIT/GNU Scheme has no such way to generate unique symbols.
- The `named` option may optionally take an argument, which is normally the name of a variable (any expression may be used, but it is evaluated whenever the tag name is needed). If used, structure instances will be tagged with that variable's value. The variable must be defined when `define-structure` is evaluated.
- The `type` option is restricted to the values `vector` and `list`.
- The `include` option is not implemented.

2.11 Macros

(This section is largely taken from the *Revised^4 Report on the Algorithmic Language Scheme*. The section on Syntactic Closures is derived from a document written by Chris Hanson. The section on Explicit Renaming is derived from a document written by William Clinger.)

Scheme programs can define and use new derived expression types, called *macros*. Program-defined expression types have the syntax

```
(keyword datum ...)
```

where *keyword* is an identifier that uniquely determines the expression type. This identifier is called the *syntactic keyword*, or simply *keyword*, of the macro. The number of the *datum*s, and their syntax, depends on the expression type.

Each instance of a macro is called a *use* of the macro. The set of rules that specifies how a use of a macro is transcribed into a more primitive expression is called the *transformer* of the macro.

MIT/GNU Scheme also supports *anonymous syntactic keywords*. This means that it's not necessary to bind a macro transformer to a syntactic keyword before it is used. Instead, any macro-transformer expression can appear as the first element of a form, and the form will be expanded by the transformer.

The macro definition facility consists of these parts:

- A set of expressions used to establish that certain identifiers are macro keywords, associate them with macro transformers, and control the scope within which a macro is defined.
- A standard high-level pattern language for specifying macro transformers, introduced by the `syntax-rules` special form.
- Two non-standard low-level languages for specifying macro transformers, *syntactic closures* and *explicit renaming*.

The syntactic keyword of a macro may shadow variable bindings, and local variable bindings may shadow keyword bindings. All macros defined using the pattern language are "hygienic" and "referentially transparent" and thus preserve Scheme's lexical scoping:

- If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers.
- If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

2.11.1 Binding Constructs for Syntactic Keywords

`let-syntax`, `letrec-syntax`, `let*-syntax` and `define-syntax` are analogous to `let`, `letrec`, `let*` and `define`, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values.

Any argument named *transformer-spec* must be a macro-transformer expression, which is one of the following:

- A macro transformer defined by the pattern language and denoted by the syntactic keyword `syntax-rules`.
- A macro transformer defined by one of the low-level mechanisms and denoted by one of the syntactic keywords `sc-macro-transformer`, `rsc-macro-transformer`, or `er-macro-transformer`.
- A syntactic keyword bound in the enclosing environment. This is used to bind another name to an existing macro transformer.

`let-syntax` *bindings expression expression ...* [standard special form]

Bindings should have the form

```
((keyword transformer-spec) ...)
```

Each *keyword* is an identifier, each *transformer-spec* is a a macro-transformer expression, and the body is a sequence of one or more expressions. It is an error for a *keyword* to appear more than once in the list of keywords being bound.

The *expression*s are expanded in the syntactic environment obtained by extending the syntactic environment of the `let-syntax` expression with macros whose keywords are the *keyword*s, bound to the specified transformers. Each binding of a *keyword* has the *expression*s as its region.

```
(let-syntax ((when (syntax-rules ()
                     ((when test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (when if (set! if 'now))
    if))                              ⇒  now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m))))                          ⇒  outer
```

`letrec-syntax` *bindings expression expression ...* [standard special form]

The syntax of `letrec-syntax` is the same as for `let-syntax`.

The *expression*s are expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` expression with macros whose keywords are the *keyword*s, bound to the specified transformers. Each binding of a *keyword* has the *bindings* as well as the *expression*s within its region, so the transformers can transcribe expressions into uses of the macros introduced by the `letrec-syntax` expression.

```
(letrec-syntax
  ((my-or (syntax-rules ()
            ((my-or) #f)
            ((my-or e) e)
            ((my-or e1 e2 ...)
             (let ((temp e1))
               (if temp
                   temp
                   (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))        ⇒  7
```

`let*-syntax` *bindings expression expression ...* [standard special form]

The syntax of `let*-syntax` is the same as for `let-syntax`.

The *expression*s are expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` expression with macros whose keywords are the *keyword*s, bound to the specified transformers. Each binding of a *keyword* has the subsequent *bindings* as well as the *expression*s within its region. Thus

```
(let*-syntax
   ((a (syntax-rules ...))
    (b (syntax-rules ...)))
  ...)
```

is equivalent to

```
(let-syntax ((a (syntax-rules ...)))
  (let-syntax ((b (syntax-rules ...)))
    ...))
```

`define-syntax` *keyword transformer-spec* [standard special form]

Keyword is an identifier, and *transformer-spec* is a macro transformer expression. The syntactic environment is extended by binding the *keyword* to the specified transformer.

The region of the binding introduced by `define-syntax` is the entire block in which it appears. However, the *keyword* may only be used after it has been defined.

MIT/GNU Scheme permits `define-syntax` to appear both at top level and within `lambda` bodies. The Revised^4 Report permits only top-level uses of `define-syntax`.

When compiling a program, a top-level instance of `define-syntax` both defines the syntactic keyword and generates code that will redefine the keyword when the program is loaded. This means that the same syntax can be used for defining macros that will be used during compilation and for defining macros to be used at run time.

Although macros may expand into definitions and syntax definitions in any context that permits them, it is an error for a definition or syntax definition to shadow a syntactic keyword whose meaning is needed to determine whether some form in the group of forms that contains the shadowing definition is in fact a definition, or, for internal definitions, is needed to determine the boundary between the group and the expressions that follow the group. For example, the following are errors:

```
(define define 3)

(begin (define begin list))

(let-syntax
  ((foo (syntax-rules ()
          ((foo (proc args ...) body ...)
           (define proc
             (lambda (args ...)
               body ...))))))
  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x)))
```

2.11.2 Pattern Language

MIT/GNU Scheme supports a high-level pattern language for specifying macro transformers. This pattern language is defined by the Revised^4 Report and is portable to other conforming Scheme implementations. To use the pattern language, specify a *transformer-spec* as a `syntax-rules` form:

`syntax-rules` [*ellipsis*] *literals syntax-rule* ... [standard special form]

Ellipsis is an identifier, and if omitted defaults to `...`. *Literals* is a list of identifiers and each *syntax-rule* should be of the form

```
(pattern template)
```

The *pattern* in a *syntax-rule* is a list *pattern* that begins with the keyword for the macro.

A *pattern* is either an identifier, a constant, or one of the following

```
(pattern ...)
(pattern pattern ... . pattern)
(pattern ... pattern ellipsis)
```

and a template is either an identifier, a constant, or one of the following

```
(element ...)
(element element ... . template)
```

where an *element* is a *template* optionally followed by an *ellipsis* and an *ellipsis* is the identifier '`...`' (which cannot be used as an identifier in either a template or a pattern).

An instance of `syntax-rules` produces a new macro transformer by specifying a sequence of hygienic rewrite rules. A use of a macro whose keyword is associated with

a transformer specified by `syntax-rules` is matched against the patterns contained in the *syntax-rule*s, beginning with the leftmost *syntax-rule*. When a match is found, the macro use is transcribed hygienically according to the template.

An identifier that appears in the pattern of a *syntax-rule* is a *pattern-variable*, unless it is the keyword that begins the pattern, is listed in *literals*, or is the identifier '`...`'. Pattern variables match arbitrary input elements and are used to refer to elements of the input in the template. It is an error for the same pattern variable to appear more than once in a *pattern*.

The keyword at the beginning of the pattern in a *syntax-rule* is not involved in the matching and is not considered a pattern variable or literal identifier.

Identifiers that appear in *literals* are interpreted as literal identifiers to be matched against corresponding subforms of the input. A subform in the input matches a literal identifier if and only if it is an identifier and either both its occurrence in the macro expression and its occurrence in the macro definition have the same lexical binding, or the two identifiers are equal and both have no lexical binding.

A subpattern followed by '`...`' can match zero or more elements of the input. It is an error for '`...`' to appear in *literals*. Within a pattern the identifier '`...`' must follow the last element of a nonempty sequence of subpatterns.

More formally, an input form F matches a pattern P if and only if:

- P is a non-literal identifier; or
- P is a literal identifier and F is an identifier with the same binding; or
- P is a list `(P_1 ... P_n)` and F is a list of n forms that match P_1 through P_n, respectively; or
- P is an improper list `(P_1 P_2 ... P_n . P_n+1)` and F is a list or improper list of n or more forms that match P_1 through P_n, respectively, and whose nth "cdr" matches P_n+1; or
- P is of the form `(P_1 ... P_n P_n+1 ellipsis)` where *ellipsis* is the identifier '`...`' and F is a proper list of at least n forms, the first n of which match P_1 through P_n, respectively, and each remaining element of F matches P_n+1; or
- P is a datum and F is equal to P in the sense of the `equal?` procedure.

It is an error to use a macro keyword, within the scope of its binding, in an expression that does not match any of the patterns.

When a macro use is transcribed according to the template of the matching *syntax rule*, pattern variables that occur in the template are replaced by the subforms they match in the input. Pattern variables that occur in subpatterns followed by one or more instances of the identifier '`...`' are allowed only in subtemplates that are followed by as many instances of '`...`'. They are replaced in the output by all of the subforms they match in the input, distributed as indicated. It is an error if the output cannot be built up as specified.

Identifiers that appear in the template but are not pattern variables or the identifier '`...`' are inserted into the output as literal identifiers. If a literal identifier is inserted as a free identifier then it refers to the binding of that identifier within whose scope

the instance of `syntax-rules` appears. If a literal identifier is inserted as a bound identifier then it is in effect renamed to prevent inadvertent captures of free identifiers.

```
(let ((=> #f))
  (cond (#t => 'ok)))           ⇒ ok
```

The macro transformer for `cond` recognizes `=>` as a local variable, and hence an expression, and not as the top-level identifier `=>`, which the macro transformer treats as a syntactic keyword. Thus the example expands into

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

instead of

```
(let ((=> #f))
  (let ((temp #t))
    (if temp
        ('ok temp))))
```

which would result in an invalid procedure call.

2.11.3 Syntactic Closures

MIT/GNU Scheme's syntax-transformation engine is an implementation of *syntactic closures*, a mechanism invented by Alan Bawden and Jonathan Rees. The main feature of the syntactic-closures mechanism is its simplicity and its close relationship to the environment models commonly used with Scheme. Using the mechanism to write macro transformers is somewhat cumbersome and can be confusing for the newly initiated, but it is easily mastered.

2.11.3.1 Syntax Terminology

This section defines the concepts and data types used by the syntactic closures facility.

- *Forms* are the syntactic entities out of which programs are recursively constructed. A form is any expression, any definition, any syntactic keyword, or any syntactic closure. The variable name that appears in a `set!` special form is also a form. Examples of forms:

  ```
  17
  #t
  car
  (+ x 4)
  (lambda (x) x)
  (define pi 3.14159)
  if
  define
  ```

- An *alias* is an alternate name for a given symbol. It can appear anywhere in a form that the symbol could be used, and when quoted it is replaced by the symbol; however, it does not satisfy the predicate `symbol?`. Macro transformers rarely distinguish symbols from aliases, referring to both as *identifiers*. Another name for an alias is *synthetic identifier*; this document uses both names.
- A *syntactic environment* maps identifiers to their meanings. More precisely, it determines whether an identifier is a syntactic keyword or a variable. If it is a keyword,

the meaning is an interpretation for the form in which that keyword appears. If it is a variable, the meaning identifies which binding of that variable is referenced. In short, syntactic environments contain all of the contextual information necessary for interpreting the meaning of a particular form.

- A *syntactic closure* consists of a form, a syntactic environment, and a list of identifiers. All identifiers in the form take their meaning from the syntactic environment, except those in the given list. The identifiers in the list are to have their meanings determined later.

  A syntactic closure may be used in any context in which its form could have been used. Since a syntactic closure is also a form, it may not be used in contexts where a form would be illegal. For example, a form may not appear as a clause in the `cond` special form.

  A syntactic closure appearing in a quoted structure is replaced by its form.

2.11.3.2 Transformer Definition

This section describes the special forms for defining syntactic-closures macro transformers, and the associated procedures for manipulating syntactic closures and syntactic environments.

`sc-macro-transformer` *expression* [special form]

The *expression* is expanded in the syntactic environment of the `sc-macro-transformer` expression, and the expanded expression is evaluated in the transformer environment to yield a macro transformer as described below. This macro transformer is bound to a macro keyword by the special form in which the `transformer` expression appears (for example, `let-syntax`).

In the syntactic closures facility, a *macro transformer* is a procedure that takes two arguments, a form and a syntactic environment, and returns a new form. The first argument, the *input form*, is the form in which the macro keyword occurred. The second argument, the *usage environment*, is the syntactic environment in which the input form occurred. The result of the transformer, the *output form*, is automatically closed in the *transformer environment*, which is the syntactic environment in which the `transformer` expression occurred.

For example, here is a definition of a `push` macro using `syntax-rules`:

```
(define-syntax push
  (syntax-rules ()
    ((push item list)
     (set! list (cons item list)))))
```

Here is an equivalent definition using `sc-macro-transformer`:

```
(define-syntax push
  (sc-macro-transformer
   (lambda (exp env)
     (let ((item (make-syntactic-closure env '() (cadr exp)))
           (list (make-syntactic-closure env '() (caddr exp))))
       `(set! ,list (cons ,item ,list))))))
```

In this example, the identifiers `set!` and `cons` are closed in the transformer environment, and thus will not be affected by the meanings of those identifiers in the usage environment `env`.

Some macros may be non-hygienic by design. For example, the following defines a `loop` macro that implicitly binds `exit` to an escape procedure. The binding of `exit` is intended to capture free references to `exit` in the body of the loop, so `exit` must be left free when the body is closed:

```
(define-syntax loop
  (sc-macro-transformer
   (lambda (exp env)
     (let ((body (cdr exp)))
       `(call-with-current-continuation
         (lambda (exit)
           (let f ()
             ,@(map (lambda (exp)
                      (make-syntactic-closure env '(exit)
                        exp))
                    body)
             (f))))))))
```

`rsc-macro-transformer` *expression* [special form]

This form is an alternative way to define a syntactic-closures macro transformer. Its syntax and usage are identical to `sc-macro-transformer`, except that the roles of the usage environment and transformer environment are reversed. (Hence RSC stands for *Reversed Syntactic Closures*.) In other words, the procedure specified by *expression* still accepts two arguments, but its second argument will be the transformer environment rather than the usage environment, and the returned expression is closed in the usage environment rather than the transformer environment.

The advantage of this arrangement is that it allows a simpler definition style in some situations. For example, here is the `push` macro from above, rewritten in this style:

```
(define-syntax push
  (rsc-macro-transformer
   (lambda (exp env)
     `(,(make-syntactic-closure env '() 'SET!)
       ,(caddr exp)
       (,(make-syntactic-closure env '() 'CONS)
        ,(cadr exp)
        ,(caddr exp))))))
```

In this style only the introduced keywords are closed, while everything else remains open.

Note that `rsc-macro-transformer` and `sc-macro-transformer` are easily interchangeable. Here is how to emulate `rsc-macro-transformer` using `sc-macro-transformer`. (This technique can be used to effect the opposite emulation as well.)

```
(define-syntax push
  (sc-macro-transformer
   (lambda (exp usage-env)
     (capture-syntactic-environment
      (lambda (env)
        (make-syntactic-closure usage-env '()
          `(,(make-syntactic-closure env '() 'SET!)
            ,(caddr exp)
            (,(make-syntactic-closure env '() 'CONS)
             ,(cadr exp)
             ,(caddr exp)))))))))
```

To assign meanings to the identifiers in a form, use `make-syntactic-closure` to close the form in a syntactic environment.

`make-syntactic-closure` *environment free-names form* [procedure]

Environment must be a syntactic environment, *free-names* must be a list of identifiers, and *form* must be a form. `make-syntactic-closure` constructs and returns a syntactic closure of *form* in *environment*, which can be used anywhere that *form* could have been used. All the identifiers used in *form*, except those explicitly excepted by *free-names*, obtain their meanings from *environment*.

Here is an example where *free-names* is something other than the empty list. It is instructive to compare the use of *free-names* in this example with its use in the `loop` example above: the examples are similar except for the source of the identifier being left free.

```
(define-syntax let1
  (sc-macro-transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (exp (cadddr exp)))
       `((lambda (,id)
           ,(make-syntactic-closure env (list id) exp))
         ,(make-syntactic-closure env '() init))))))
```

`let1` is a simplified version of `let` that only binds a single identifier, and whose body consists of a single expression. When the body expression is syntactically closed in its original syntactic environment, the identifier that is to be bound by `let1` must be left free, so that it can be properly captured by the `lambda` in the output form.

In most situations, the *free-names* argument to `make-syntactic-closure` is the empty list. In those cases, the more succinct `close-syntax` can be used:

`close-syntax` *form environment* [procedure]

Environment must be a syntactic environment and *form* must be a form. Returns a new syntactic closure of *form* in *environment*, with no free names. Entirely equivalent to

```
(make-syntactic-closure environment '() form)
```

To obtain a syntactic environment other than the usage environment, use `capture-syntactic-environment`.

`capture-syntactic-environment` *procedure* [procedure]

`capture-syntactic-environment` returns a form that will, when transformed, call *procedure* on the current syntactic environment. *Procedure* should compute and return a new form to be transformed, in that same syntactic environment, in place of the form.

An example will make this clear. Suppose we wanted to define a simple `loop-until` keyword equivalent to

```
(define-syntax loop-until
  (syntax-rules ()
    ((loop-until id init test return step)
     (letrec ((loop
               (lambda (id)
                 (if test return (loop step)))))
       (loop init)))))
```

The following attempt at defining `loop-until` has a subtle bug:

```
(define-syntax loop-until
  (sc-macro-transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
       `(letrec ((loop
                  (lambda (,id)
                    (if ,(close test (list id))
                        ,(close return (list id))
                        (loop ,(close step (list id)))))))
          (loop ,(close init '()))))))))
```

This definition appears to take all of the proper precautions to prevent unintended captures. It carefully closes the subexpressions in their original syntactic environment and it leaves the `id` identifier free in the `test`, `return`, and `step` expressions, so that it will be captured by the binding introduced by the `lambda` expression. Unfortunately it uses the identifiers `if` and `loop` *within* that `lambda` expression, so if the user of `loop-until` just happens to use, say, `if` for the identifier, it will be inadvertently captured.

The syntactic environment that `if` and `loop` want to be exposed to is the one just outside the `lambda` expression: before the user's identifier is added to the syntactic environment, but after the identifier `loop` has been added. `capture-syntactic-environment` captures exactly that environment as follows:

```
(define-syntax loop-until
  (sc-macro-transformer
   (lambda (exp env)
     (let ((id (cadr exp))
           (init (caddr exp))
           (test (cadddr exp))
           (return (cadddr (cdr exp)))
           (step (cadddr (cddr exp)))
           (close
            (lambda (exp free)
              (make-syntactic-closure env free exp))))
       `(letrec ((loop
                  ,(capture-syntactic-environment
                    (lambda (env)
                      `(lambda (,id)
                         (,(make-syntactic-closure env '() `if)
                          ,(close test (list id))
                          ,(close return (list id))
                          (,(make-syntactic-closure env '() `loop)
                           ,(close step (list id)))))))))
          (loop ,(close init '())))))))
```

In this case, having captured the desired syntactic environment, it is convenient to construct syntactic closures of the identifiers `if` and the `loop` and use them in the body of the `lambda`.

A common use of `capture-syntactic-environment` is to get the transformer environment of a macro transformer:

```
(sc-macro-transformer
 (lambda (exp env)
   (capture-syntactic-environment
    (lambda (transformer-env)
      ...))))
```

2.11.3.3 Identifiers

This section describes the procedures that create and manipulate identifiers. The identifier data type extends the syntactic closures facility to be compatible with the high-level `syntax-rules` facility.

As discussed earlier, an identifier is either a symbol or an *alias*. An alias is implemented as a syntactic closure whose *form* is an identifier:

`(make-syntactic-closure env '() 'a)` ⇒ an alias

Aliases are implemented as syntactic closures because they behave just like syntactic closures most of the time. The difference is that an alias may be bound to a new value (for example by `lambda` or `let-syntax`); other syntactic closures may not be used this way. If an alias is bound, then within the scope of that binding it is looked up in the syntactic environment just like any other identifier.

Aliases are used in the implementation of the high-level facility `syntax-rules`. A macro transformer created by `syntax-rules` uses a template to generate its output form, substituting subforms of the input form into the template. In a syntactic closures implementation, all of the symbols in the template are replaced by aliases closed in the transformer environment, while the output form itself is closed in the usage environment. This guarantees that the macro transformation is hygienic, without requiring the transformer to know the syntactic roles of the substituted input subforms.

`identifier?` *object* [procedure]

Returns `#t` if *object* is an identifier, otherwise returns `#f`. Examples:

```
(identifier? 'a)        ⇒ #t
(identifier? (make-syntactic-closure env '() 'a))
                        ⇒ #t

(identifier? "a")       ⇒ #f
(identifier? #\a)       ⇒ #f
(identifier? 97)        ⇒ #f
(identifier? #f)        ⇒ #f
(identifier? '(a))      ⇒ #f
(identifier? '#(a))     ⇒ #f
```

The predicate `eq?` is used to determine if two identifers are "the same". Thus `eq?` can be used to compare identifiers exactly as it would be used to compare symbols. Often, though, it is useful to know whether two identifiers "mean the same thing". For example, the `cond` macro uses the symbol `else` to identify the final clause in the conditional. A macro transformer for `cond` cannot just look for the symbol `else`, because the `cond` form might be the output of another macro transformer that replaced the symbol `else` with an alias. Instead the transformer must look for an identifier that "means the same thing" in the usage environment as the symbol `else` means in the transformer environment.

`identifier=?` *environment1 identifier1 environment2 identifier2* [procedure]

Environment1 and *environment2* must be syntactic environments, and *identifier1* and *identifier2* must be identifiers. `identifier=?` returns `#t` if the meaning of *identifier1* in *environment1* is the same as that of *identifier2* in *environment2*, otherwise it returns `#f`. Examples:

```
(let-syntax
    ((foo
      (sc-macro-transformer
       (lambda (form env)
         (capture-syntactic-environment
          (lambda (transformer-env)
            (identifier=? transformer-env 'x env 'x)))))))
  (list (foo)
        (let ((x 3))
          (foo))))
                        ⇒ (#t #f)
```

```
(let-syntax ((bar foo))
  (let-syntax
      ((foo
        (sc-macro-transformer
         (lambda (form env)
           (capture-syntactic-environment
            (lambda (transformer-env)
              (identifier=? transformer-env 'foo
                            env (cadr form))))))))
    (list (foo foo)
          (foo bar))))
                          ⇒ (#f #t)
```

Sometimes it is useful to be able to introduce a new identifier that is guaranteed to be different from any existing identifier, similarly to the way that `generate-uninterned-symbol` is used.

`make-synthetic-identifier` *identifier* [procedure]

Creates and returns and new synthetic identifier (alias) that is guaranteed to be different from all existing identifiers. *Identifier* is any existing identifier, which is used in deriving the name of the new identifier.

2.11.4 Explicit Renaming

Explicit renaming is an alternative facility for defining macro transformers. In the MIT/GNU Scheme implementation, explicit-renaming transformers are implemented as an abstraction layer on top of syntactic closures. An explicit-renaming macro transformer is defined by an instance of the `er-macro-transformer` keyword:

`er-macro-transformer` *expression* [special form]

The *expression* is expanded in the syntactic environment of the `er-macro-transformer` expression, and the expanded expression is evaluated in the transformer environment to yield a macro transformer as described below. This macro transformer is bound to a macro keyword by the special form in which the `transformer` expression appears (for example, `let-syntax`).

In the explicit-renaming facility, a *macro transformer* is a procedure that takes three arguments, a form, a renaming procedure, and a comparison predicate, and returns a new form. The first argument, the *input form*, is the form in which the macro keyword occurred.

The second argument to a transformation procedure is a *renaming procedure* that takes the representation of an identifier as its argument and returns the representation of a fresh identifier that occurs nowhere else in the program. For example, the transformation procedure for a simplified version of the `let` macro might be written as

```
(lambda (exp rename compare)
  (let ((vars (map car (cadr exp)))
        (inits (map cadr (cadr exp)))
        (body (cddr exp)))
    `((lambda ,vars ,@body)
      ,@inits)))
```

This would not be hygienic, however. A hygienic `let` macro must rename the identifier `lambda` to protect it from being captured by a local binding. The renaming effectively creates an fresh alias for `lambda`, one that cannot be captured by any subsequent binding:

```
(lambda (exp rename compare)
  (let ((vars (map car (cadr exp)))
        (inits (map cadr (cadr exp)))
        (body (cddr exp)))
    `((,(rename 'lambda) ,vars ,@body)
      ,@inits)))
```

The expression returned by the transformation procedure will be expanded in the syntactic environment obtained from the syntactic environment of the macro application by binding any fresh identifiers generated by the renaming procedure to the denotations of the original identifiers in the syntactic environment in which the macro was defined. This means that a renamed identifier will denote the same thing as the original identifier unless the transformation procedure that renamed the identifier placed an occurrence of it in a binding position.

The renaming procedure acts as a mathematical function in the sense that the identifiers obtained from any two calls with the same argument will be the same in the sense of `eqv?`. It is an error if the renaming procedure is called after the transformation procedure has returned.

The third argument to a transformation procedure is a *comparison predicate* that takes the representations of two identifiers as its arguments and returns true if and only if they denote the same thing in the syntactic environment that will be used to expand the transformed macro application. For example, the transformation procedure for a simplified version of the `cond` macro can be written as

```
(lambda (exp rename compare)
  (let ((clauses (cdr exp)))
    (if (null? clauses)
        `(,(rename 'quote) unspecified)
        (let* ((first (car clauses))
               (rest (cdr clauses))
               (test (car first)))
          (cond ((and (identifier? test)
                      (compare test (rename 'else)))
                 `(,(rename 'begin) ,@(cdr first)))
                (else `(,(rename 'if)
                        ,test
                        (,(rename 'begin) ,@(cdr first))
                        (cond ,@rest)))))))))
```

In this example the identifier `else` is renamed before being passed to the comparison predicate, so the comparison will be true if and only if the test expression is an identifier that denotes the same thing in the syntactic environment of the expression being transformed as `else` denotes in the syntactic environment in which the `cond` macro was defined. If `else` were not renamed before being passed to the comparison predicate, then it would match a local variable that happened to be named `else`, and the macro would not be hygienic.

Some macros are non-hygienic by design. For example, the following defines a `loop` macro that implicitly binds `exit` to an escape procedure. The binding of `exit` is intended to capture free references to `exit` in the body of the loop, so `exit` is not renamed.

```
(define-syntax loop
  (er-macro-transformer
   (lambda (x r c)
     (let ((body (cdr x)))
       `(,(r 'call-with-current-continuation)
         (,(r 'lambda) (exit)
          (,(r 'let) ,(r 'f) () ,@body (,(r 'f)))))))))
```

Suppose a `while` macro is implemented using `loop`, with the intent that `exit` may be used to escape from the `while` loop. The `while` macro cannot be written as

```
(define-syntax while
  (syntax-rules ()
    ((while test body ...)
     (loop (if (not test) (exit #f))
           body ...))))
```

because the reference to `exit` that is inserted by the `while` macro is intended to be captured by the binding of `exit` that will be inserted by the `loop` macro. In other words, this `while` macro is not hygienic. Like `loop`, it must be written using the `er-macro-transformer` syntax:

```
(define-syntax while
  (er-macro-transformer
   (lambda (x r c)
     (let ((test (cadr x))
           (body (cddr x)))
       `(,(r 'loop)
         (,(r 'if) (,(r 'not) ,test) (exit #f))
         ,@body)))))
```

2.12 SRFI syntax

Several special forms have been introduced to support some of the Scheme Requests for Implementation (SRFI). Note that MIT/GNU Scheme has for some time supported SRFI 23 (error-reporting mechanism) and SRFI 30 (nested multi-line comments), since these SRFIs reflect existing practice rather than introducing new functionality.

2.12.1 cond-expand (SRFI 0)

SRFI 0 is a mechanism for portably determining the availability of SRFI *features*. The `cond-expand` special form conditionally expands according to the features available.

`cond-expand` *clause clause dots* [standard special form]

Each *clause* has the form

```
(feature-requirement expression ...)
```

where *feature-requirement* can have one of the following forms:

```
feature-identifier
(and feature-requirement ...)
(or feature-requirement ...)
(not feature-requirement)
else
```

(Note that at most one `else` clause may be present, and it must always be the last clause.)

The `cond-expand` special form tests for the existence of features at macro-expansion time. It either expands into the body of one of its *clause*s or signals an error during syntactic processing. `cond-expand` expands into the body of the first *clause* whose *feature-requirement* is currently satisfied (an `else` *clause*, if present, is selected if none of the previous *clauses* is selected).

A *feature-requirement* has an obvious interpretation as a logical formula, where the *feature-identifier* variables have meaning true if the feature corresponding to the *feature-identifier*, as specified in the SRFI registry, is in effect at the location of the `cond-expand` form, and false otherwise. A *feature-requirement* is satisfied if its formula is true under this interpretation.

```
(cond-expand
  ((and srfi-1 srfi-10)
   (write 1))
  ((or srfi-1 srfi-10)
   (write 2))
  (else))

(cond-expand
  (command-line
   (define (program-name) (car (argv)))))
```

The second example assumes that `command-line` is an alias for some feature which gives access to command line arguments. Note that an error will be signaled at macro-expansion time if this feature is not present.

Note that MIT/GNU Scheme allows `cond-expand` in any context where a special form is allowed. This is an extension of the semantics defined by SRFI 0, which only allows `cond-expand` at top level.

2.12.2 receive (SRFI 8)

SRFI 8 defines a convenient syntax to bind an identifier to each of the values of a multiple-valued expression and then evaluate an expression in the scope of the bindings. As an instance of this pattern, consider the following excerpt from a '`quicksort`' procedure:

```
(call-with-values
  (lambda ()
    (partition (precedes pivot) others))
  (lambda (fore aft)
    (append (qsort fore) (cons pivot (qsort aft)))))
```

Here '`partition`' is a multiple-valued procedure that takes two arguments, a predicate and a list, and returns two lists, one comprising the list elements that satisfy the predicate, the other those that do not. The purpose of the expression shown is to partition the list '`others`', sort each of the sublists, and recombine the results into a sorted list.

For our purposes, the important step is the binding of the identifiers '`fore`' and '`aft`' to the values returned by '`partition`'. Expressing the construction and use of these bindings with the call-by-values primitive is cumbersome: One must explicitly embed the expression that provides the values for the bindings in a parameterless procedure, and one must explicitly embed the expression to be evaluated in the scope of those bindings in another procedure, writing as its parameters the identifiers that are to be bound to the values received.

These embeddings are boilerplate, exposing the underlying binding mechanism but not revealing anything relevant to the particular program in which it occurs. So the use of a syntactic abstraction that exposes only the interesting parts – the identifiers to be bound, the multiple-valued expression that supplies the values, and the body of the receiving procedure – makes the code more concise and more readable:

```
(receive (fore aft) (partition (precedes pivot) others)
  (append (qsort fore) (cons pivot (qsort aft))))
```

The advantages are similar to those of a '`let`' expression over a procedure call with a '`lambda`' expression as its operator. In both cases, cleanly separating a "header" in which the bindings are established from a "body" in which they are used makes it easier to follow the code.

`receive` *formals expression body* [special form]

Formals and *body* are defined as for '`lambda`' (see Section 2.1 [Lambda Expressions], page 15). Specifically, *formals* can have the following forms (the use of '`#!optional`' and '`#!rest`' is also allowed in *formals* but is omitted for brevity):

'`(`*ident1 ... identN*`)`'
: The environment in which the '`receive`' expression is evaluated is extended by binding *ident1*, ..., *identN* to fresh locations. The *expression* is evaluated, and its values are stored into those locations. (It is an error if *expression* does not have exactly N values.)

'*ident*'
: The environment in which the '`receive`' expression is evaluated is extended by binding *ident* to a fresh location. The *expression* is evaluated, its values are converted into a newly allocated list, and the list is stored in the location bound to *ident*.

‘(*ident1 ... identN . identN+1*)’
: The environment in which the ‘`receive`’ expression is evaluated is extended by binding *ident1*, ..., *identN+1* to fresh locations. The *expression* is evaluated. Its first N values are stored into the locations bound to *ident1* ... *identN*. Any remaining values are converted into a newly allocated list, which is stored into the location bound to *identN+1*. (It is an error if *expression* does not have at least N values.)

In any case, the expressions in *body* are evaluated sequentially in the extended environment. The results of the last expression in the body are the values of the ‘`receive`’ expression.

2.12.3 and-let* (SRFI 2)

SRFI 2 provides a form that combines ‘`and`’ and ‘`let*`’ for a logically short-circuiting sequential binding operator.

`and-let*` (*clause ...*) *body* [special form]
: Runs through each of the clauses left-to-right, short-circuiting like ‘`and`’ in that the first false clause will result in the whole ‘`and-let*`’ form returning false. If a body is supplied, and all of the clauses evaluate true, then the body is evaluated sequentially as if in a ‘`begin`’ form, and the value of the ‘`and-let*`’ expression is the value of the last body form, evaluated in a tail position with respect to the ‘`and-let*`’ expression. If no body is supplied, the value of the last clause, also evaluated in a tail position with respect to the ‘`and-let*`’ expression, is used instead.

  Each *clause* should have one of the following forms:

  ‘*identifier*’
  : in which case *identifier*’s value is tested.

  ‘(*expression*)’
  : in which case the value of *expression* is tested.

  ‘(*identifier expression*)’
  : in which case *expression* is evaluated, and, if its value is not false, *identifier* is bound to that value for the remainder of the clauses and the optional body.

Example:

```
(and-let* ((list (compute-list))
           ((pair? list))
           (item (car list))
           ((integer? item)))
  (sqrt item))
```

2.12.4 define-record-type (SRFI 9)

The ‘`define-record-type`’ syntax described in SRFI 9 is a slight simplification of one written for Scheme 48 by Jonathan Rees. Unlike many record-defining special forms, it does not create any new identifiers. Instead, the names of the record type, predicate, constructor, and so on are all listed explicitly in the source. This has the following advantages:

- It can be defined using a simple macro in Scheme implementations that provide a procedural interface for creating record types.
- It does not restrict users to a particular naming convention.
- Tools like `grep` and the GNU Emacs tag facility will see the defining occurrence of each identifier.

`define-record-type` *type-name* [extended standard special form]
(*constructor-name field-tag ...*) *predicate-name field-spec ...*

Type-name, *contructor-name*, *field-tag*, and *predicate-name* are identifiers. *Field-spec* has one of these two forms:

```
(field-tag accessor-name)
(field-tag accessor-name modifier-name)
```

where *field-tag*, *accessor-name*, and *modifier-name* are each identifiers.

`define-record-type` is generative: each use creates a new record type that is distinct from all existing types, including other record types and Scheme's predefined types. Record-type definitions may only occur at top-level (there are two possible semantics for "internal" record-type definitions, generative and nongenerative, and no consensus as to which is better).

An instance of `define-record-type` is equivalent to the following definitions:

- *Type-name* is bound to a representation of the record type itself. Operations on record types, such as defining print methods, reflection, etc. are left to other SRFIs.
- *constructor-name* is bound to a procedure that takes as many arguments as there are *field-tag*s in the (*constructor-name ...*) subform and returns a new *type-name* record. Fields whose tags are listed with *constructor-name* have the corresponding argument as their initial value. The initial values of all other fields are unspecified.
- *predicate-name* is a predicate that returns `#t` when given a value returned by *constructor-name* and `#f` for everything else.
- Each *accessor-name* is a procedure that takes a record of type *type-name* and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each *modifier-name* is a procedure that takes a record of type *type-name* and a value which becomes the new value of the corresponding field; an unspecified value is returned. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

Assigning the value of any of these identifiers has no effect on the behavior of any of their original values.

The following

```
(define-record-type :pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

defines ‘kons’ to be a constructor, ‘kar’ and ‘kdr’ to be accessors, ‘set-kar!’ to be a modifier, and ‘pare?’ to be a predicate for objects of type ‘:pare’.

```
(pare? (kons 1 2))          ⇒ #t
(pare? (cons 1 2))          ⇒ #f
(kar (kons 1 2))            ⇒ 1
(kdr (kons 1 2))            ⇒ 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k))                  ⇒ 3
```

3 Equivalence Predicates

A *predicate* is a procedure that always returns a boolean value (`#t` or `#f`). An *equivalence predicate* is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive). Of the equivalence predicates described in this section, `eq?` is the finest or most discriminating, and `equal?` is the coarsest. `eqv?` is slightly less discriminating than `eq?`.

`eqv?` *obj1 obj2* [procedure]

The `eqv?` procedure defines a useful equivalence relation on objects. Briefly, it returns `#t` if *obj1* and *obj2* should normally be regarded as the same object.

The `eqv?` procedure returns `#t` if:

- *obj1* and *obj2* are both `#t` or both `#f`.
- *obj1* and *obj2* are both interned symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
    ⇒ #t
```

- *obj1* and *obj2* are both numbers, are numerically equal according to the `=` procedure, and are either both exact or both inexact (see Chapter 4 [Numbers], page 63).
- *obj1* and *obj2* are both characters and are the same character according to the `char=?` procedure (see Chapter 5 [Characters], page 109).
- both *obj1* and *obj2* are the empty list.
- *obj1* and *obj2* are procedures whose location tags are equal.
- *obj1* and *obj2* are pairs, vectors, strings, bit strings, records, cells, or weak pairs that denote the same locations in the store.

The `eqv?` procedure returns `#f` if:

- *obj1* and *obj2* are of different types.
- one of *obj1* and *obj2* is `#t` but the other is `#f`.
- *obj1* and *obj2* are symbols but

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
    ⇒ #f
```

- one of *obj1* and *obj2* is an exact number but the other is an inexact number.
- *obj1* and *obj2* are numbers for which the `=` procedure returns `#f`.
- *obj1* and *obj2* are characters for which the `char=?` procedure returns `#f`.
- one of *obj1* and *obj2* is the empty list but the other is not.
- *obj1* and *obj2* are procedures that would behave differently (return a different value or have different side effects) for some arguments.
- *obj1* and *obj2* are pairs, vectors, strings, bit strings, records, cells, or weak pairs that denote distinct locations.

Some examples:

```
(eqv? 'a 'a)                    ⇒  #t
(eqv? 'a 'b)                    ⇒  #f
(eqv? 2 2)                      ⇒  #t
(eqv? '() '())                  ⇒  #t
(eqv? 100000000 100000000)      ⇒  #t
(eqv? (cons 1 2) (cons 1 2))    ⇒  #f
(eqv? (lambda () 1)
      (lambda () 2))            ⇒  #f
(eqv? #f 'nil)                  ⇒  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                   ⇒  #t
```

The following examples illustrate cases in which the above rules do not fully specify the behavior of `eqv?`. All that can be said about such cases is that the value returned by `eqv?` must be a boolean.

```
(eqv? "" "")                    ⇒  unspecified
(eqv? '#() '#())                ⇒  unspecified
(eqv? (lambda (x) x)
      (lambda (x) x))           ⇒  unspecified
(eqv? (lambda (x) x)
      (lambda (y) y))           ⇒  unspecified
```

The next set of examples shows the use of `eqv?` with procedures that have local state. `gen-counter` must return a distinct procedure every time, since each procedure has its own internal counter. `gen-loser`, however, returns equivalent procedures each time, since the local state does not affect the value or side effects of the procedures.

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                   ⇒  #t
(eqv? (gen-counter) (gen-counter))
                                ⇒  #f

(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                   ⇒  #t
(eqv? (gen-loser) (gen-loser))
                                ⇒  unspecified
```

```
(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                                    ⇒  unspecified

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
                                    ⇒  #f
```

Objects of distinct types must never be regarded as the same object.

Since it is an error to modify constant objects (those returned by literal expressions), the implementation may share structure between constants where appropriate. Thus the value of `eqv?` on constants is sometimes unspecified.

```
(let ((x '(a)))
  (eqv? x x))                       ⇒  #t
(eqv? '(a) '(a))                    ⇒  unspecified
(eqv? "a" "a")                      ⇒  unspecified
(eqv? '(b) (cdr '(a b)))            ⇒  unspecified
```

Rationale: The above definition of `eqv?` allows implementations latitude in their treatment of procedures and literals: implementations are free either to detect or to fail to detect that two procedures or two literals are equivalent to each other, and can decide whether or not to merge representations of equivalent objects by using the same pointer or bit pattern to represent both.

`eq?` *obj1 obj2* [procedure]

`eq?` is similar to `eqv?` except that in some cases it is capable of discerning distinctions finer than those detectable by `eqv?`.

`eq?` and `eqv?` are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, records, and non-empty strings and vectors. `eq?`'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when `eqv?` would also return true. `eq?` may also behave differently from `eqv?` on empty vectors and empty strings.

```
(eq? 'a 'a)                     ⇒  #t
(eq? '(a) '(a))                 ⇒  unspecified
(eq? (list 'a) (list 'a))       ⇒  #f
(eq? "a" "a")                   ⇒  unspecified
(eq? "" "")                     ⇒  unspecified
(eq? '() '())                   ⇒  #t
(eq? 2 2)                       ⇒  unspecified
(eq? #\A #\A)                   ⇒  unspecified
(eq? car car)                   ⇒  #t
(let ((n (+ 2 3)))
  (eq? n n))                    ⇒  unspecified
(let ((x '(a)))
  (eq? x x))                    ⇒  #t
(let ((x '#()))
  (eq? x x))                    ⇒  #t
(let ((p (lambda (x) x)))
  (eq? p p))                    ⇒  #t
```

Rationale: It will usually be possible to implement `eq?` much more efficiently than `eqv?`, for example, as a simple pointer comparison instead of as some more complicated operation. One reason is that it may not be possible to compute `eqv?` of two numbers in constant time, whereas `eq?` implemented as pointer comparison will always finish in constant time. `eq?` may be used like `eqv?` in applications using procedures to implement objects with state since it obeys the same constraints as `eqv?`.

`equal?` *obj1 obj2* [procedure]

`equal?` recursively compares the contents of pairs, vectors, and strings, applying `eqv?` on other objects such as numbers, symbols, and records. A rule of thumb is that objects are generally `equal?` if they print the same. `equal?` may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)                  ⇒  #t
(equal? '(a) '(a))              ⇒  #t
(equal? '(a (b) c)
        '(a (b) c))             ⇒  #t
(equal? "abc" "abc")            ⇒  #t
(equal? 2 2)                    ⇒  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))     ⇒  #t
(equal? (lambda (x) x)
        (lambda (y) y))         ⇒  unspecified
```

4 Numbers

(This section is largely taken from the *Revised^4 Report on the Algorithmic Language Scheme*.)

Numerical computation has traditionally been neglected by the Lisp community. Until Common Lisp there was no carefully thought out strategy for organizing numerical computation, and with the exception of the MacLisp system little effort was made to execute numerical code efficiently. This report recognizes the excellent work of the Common Lisp committee and accepts many of their recommendations. In some ways this report simplifies and generalizes their proposals in a manner consistent with the purposes of Scheme.

It is important to distinguish between the mathematical numbers, the Scheme numbers that attempt to model them, the machine representations used to implement the Scheme numbers, and notations used to write numbers. This report uses the types *number*, *complex*, *real*, *rational*, and *integer* to refer to both mathematical numbers and Scheme numbers. Machine representations such as fixed point and floating point are referred to by names such as *fixnum* and *flonum*.

4.1 Numerical types

Mathematically, numbers may be arranged into a tower of subtypes in which each level is a subset of the level above it:

number
complex
real
rational
integer

For example, 3 is an integer. Therefore 3 is also a rational, a real, and a complex. The same is true of the Scheme numbers that model 3. For Scheme numbers, these types are defined by the predicates `number?`, `complex?`, `real?`, `rational?`, and `integer?`.

There is no simple relationship between a number's type and its representation inside a computer. Although most implementations of Scheme will offer at least two different representations of 3, these different representations denote the same integer.

Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible. Although an implementation of Scheme may use fixnum, flonum, and perhaps other representations for numbers, this should not be apparent to a casual programmer writing simple programs.

It is necessary, however, to distinguish between numbers that are represented exactly and those that may not be. For example, indexes into data structures must be known exactly, as must some polynomial coefficients in a symbolic algebra system. On the other hand, the results of measurements are inherently inexact, and irrational numbers may be approximated by rational and therefore inexact approximations. In order to catch uses of inexact numbers where exact numbers are required, Scheme explicitly distinguishes exact from inexact numbers. This distinction is orthogonal to the dimension of type.

4.2 Exactness

Scheme numbers are either *exact* or *inexact*. A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is inexact if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus inexactness is a contagious property of a number.

If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results will be mathematically equivalent. This is generally not true of computations involving inexact numbers since approximate methods such as floating point arithmetic may be used, but it is the duty of each implementation to make the result as close as practical to the mathematically ideal result.

Rational operations such as `+` should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction or it may silently coerce its result to an inexact value. See Section 4.3 [Implementation restrictions], page 64.

With the exception of `exact`, the operations described in this section must generally return inexact results when given any inexact arguments. An operation may, however, return an exact result if it can prove that the value of the result is unaffected by the inexactness of its arguments. For example, multiplication of any number by an exact zero may produce an exact zero result, even if the other argument is inexact.

4.3 Implementation restrictions

Implementations of Scheme are not required to implement the whole tower of subtypes (see Section 4.1 [Numerical types], page 63), but they must implement a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language. For example, an implementation in which all numbers are real may still be quite useful.[1]

Implementations may also support only a limited range of numbers of any type, subject to the requirements of this section. The supported range for exact numbers of any type may be different from the supported range for inexact numbers of that type. For example, an implementation that uses flonums to represent all its inexact real numbers may support a practically unbounded range of exact integers and rationals while limiting the range of inexact reals (and therefore the range of inexact integers and rationals) to the dynamic range of the flonum format. Furthermore the gaps between the representable inexact integers and rationals are likely to be very large in such an implementation as the limits of this range are approached.

An implementation of Scheme must support exact integers throughout the range of numbers that may be used for indexes of lists, vectors, and strings or that may result from computing the length of a list, vector, or string. The `length`, `vector-length`, and `string-length` procedures must return an exact integer, and it is an error to use anything but an exact integer as an index. Furthermore any integer constant within the index range, if

[1] MIT/GNU Scheme implements the whole tower of numerical types. It has unlimited-precision exact integers and exact rationals. Flonums are used to implement all inexact reals; on machines that support IEEE floating-point arithmetic these are double-precision floating-point numbers.

expressed by an exact integer syntax, will indeed be read as an exact integer, regardless of any implementation restrictions that may apply outside this range. Finally, the procedures listed below will always return an exact integer result provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer within the implementation:

```
*                gcd                modulo
+                imag-part          numerator
-                exact              quotient
abs              lcm                rationalize
angle            magnitude          real-part
ceiling          make-polar         remainder
denominator      make-rectangular   round
expt             max                truncate
floor            min
```

Implementations are encouraged, but not required, to support exact integers and exact rationals of practically unlimited size and precision, and to implement the above procedures and the `/` procedure in such a way that they always return exact results when given exact arguments. If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number. Such a coercion may cause an error later.

An implementation may use floating point and other approximate representation strategies for inexact numbers. This report recommends, but does not require, that the IEEE 32-bit and 64-bit floating point standards be followed by implementations that use flonum representations, and that implementations using other representations should match or exceed the precision achievable using these floating point standards.

In particular, implementations that use flonum representations must follow these rules: A flonum result must be represented with at least as much precision as is used to express any of the inexact arguments to that operation. It is desirable (but not required) for potentially inexact operations such as `sqrt`, when applied to exact arguments, to produce exact answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If, however, an exact number is operated upon so as to produce an inexact result (as by `sqrt`), and if the result is represented as a flonum, then the most precise flonum format available must be used; but if the result is represented in some other way then the representation must have at least as much precision as the most precise flonum format available.

Although Scheme allows a variety of written notations for numbers, any particular implementation may support only some of them.[2] For example, an implementation in which all numbers are real need not support the rectangular and polar notations for complex numbers. If an implementation encounters an exact numerical constant that it cannot represent as an exact number, then it may either report a violation of an implementation restriction or it may silently represent the constant by an inexact number.

[2] MIT/GNU Scheme implements all of the written notations for numbers.

4.4 Syntax of numerical constants

A number may be written in binary, octal, decimal, or hexadecimal by the use of a radix prefix. The radix prefixes are `#b` (binary), `#o` (octal), `#d` (decimal), and `#x` (hexadecimal). With no radix prefix, a number is assumed to be expressed in decimal.

A numerical constant may be specified to be either exact or inexact by a prefix. The prefixes are `#e` for exact, and `#i` for inexact. An exactness prefix may appear before or after any radix prefix that is used. If the written representation of a number has no exactness prefix, the constant may be either inexact or exact. It is inexact if it contains a decimal point, an exponent, or a `#` character in the place of a digit, otherwise it is exact.

In systems with inexact numbers of varying precisions it may be useful to specify the precision of a constant. For this purpose, numerical constants may be written with an *exponent marker* that indicates the desired precision of the inexact representation. The letters `s`, `f`, `d`, and `l` specify the use of *short*, *single*, *double*, and *long* precision, respectively. (When fewer than four internal inexact representations exist, the four size specifications are mapped onto those available. For example, an implementation with two internal representations may map short and single together and long and double together.) In addition, the exponent marker `e` specifies the default precision for the implementation. The default precision has at least as much precision as *double*, but implementations may wish to allow this default to be set by the user.

`3.14159265358979F0`
: Round to single — `3.141593`

`0.6L0`
: Extend to long — `.600000000000000`

4.5 Numerical operations

See Section 1.1.3 [Entry Format], page 5, for a summary of the naming conventions used to specify restrictions on the types of arguments to numerical routines. The examples used in this section assume that any numerical constant written using an exact notation is indeed represented as an exact number. Some examples also assume that certain numerical constants written using an inexact notation can be represented without loss of accuracy; the inexact constants were chosen so that this is likely to be true in implementations that use flonums to represent inexact numbers.

`number?` *object* [procedure]
`complex?` *object* [procedure]
`real?` *object* [procedure]
`rational?` *object* [procedure]
`integer?` *object* [procedure]

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return `#t` if the object is of the named type, and otherwise they return `#f`. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.[3]

[3] In MIT/GNU Scheme the `rational?` procedure is the same as `real?`, and the `complex?` procedure is the same as `number?`.

If z is an inexact complex number, then `(real? z)` is true if and only if `(zero? (imag-part z))` is true. If x is an inexact real number, then `(integer? x)` is true if and only if `(= x (round x))`.

```
(complex? 3+4i)         ⇒  #t
(complex? 3)            ⇒  #t
(real? 3)               ⇒  #t
(real? -2.5+0.0i)       ⇒  #t
(real? #e1e10)          ⇒  #t
(rational? 6/10)        ⇒  #t
(rational? 6/3)         ⇒  #t
(integer? 3+0i)         ⇒  #t
(integer? 3.0)          ⇒  #t
(integer? 8/4)          ⇒  #t
```

Note: The behavior of these type predicates on inexact numbers is unreliable, since any inaccuracy may affect the result.

`exact?` *z* [procedure]
`inexact?` *z* [procedure]

These numerical predicates provide tests for the exactness of a quantity. For any Scheme number, precisely one of these predicates is true.

`exact-integer?` *object* [procedure]
`exact-nonnegative-integer?` *object* [procedure]
`exact-rational?` *object* [procedure]

These procedures test for some very common types of numbers. These tests could be written in terms of simpler predicates, but are more efficient.

`=` *z1 z2 z3 . . .* [procedure]
`<` *x1 x2 x3 . . .* [procedure]
`>` *x1 x2 x3 . . .* [procedure]
`<=` *x1 x2 x3 . . .* [procedure]
`>=` *x1 x2 x3 . . .* [procedure]

These procedures return `#t` if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

These predicates are transitive. Note that the traditional implementations of these predicates in Lisp-like languages are not transitive.

Note: While it is not an error to compare inexact numbers using these predicates, the results may be unreliable because a small inaccuracy may affect the result; this is especially true of `=` and `zero?`. When in doubt, consult a numerical analyst.

`zero?` *z* [procedure]
`positive?` *x* [procedure]
`negative?` *x* [procedure]
`odd?` *x* [procedure]
`even?` *x* [procedure]

These numerical predicates test a number for a particular property, returning `#t` or `#f`. See note above regarding inexact numbers.

`max` *x1 x2 ...* [procedure]
`min` *x1 x2 ...* [procedure]

These procedures return the maximum or minimum of their arguments.

```
(max 3 4)               ⇒  4    ; exact
(max 3.9 4)             ⇒  4.0  ; inexact
```

Note: If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If `min` or `max` is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure may report a violation of an implementation restriction.[4]

`+` *z1 ...* [procedure]
`*` *z1 ...* [procedure]

These procedures return the sum or product of their arguments.

```
(+ 3 4)                 ⇒  7
(+ 3)                   ⇒  3
(+)                     ⇒  0
(* 4)                   ⇒  4
(*)                     ⇒  1
```

`-` *z1 z2 ...* [procedure]
`/` *z1 z2 ...* [procedure]

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

```
(- 3 4)                 ⇒  -1
(- 3 4 5)               ⇒  -6
(- 3)                   ⇒  -3
(/ 3 4 5)               ⇒  3/20
(/ 3)                   ⇒  1/3
```

`1+` *z* [procedure]
`-1+` *z* [procedure]

`(1+ z)` is equivalent to `(+ z 1)`; `(-1+ z)` is equivalent to `(- z 1)`.

`abs` *x* [procedure]

Returns a real number with the magnitude of *x* and nonnegative sign bit. If *x* is a NaN, then the result has nonnegative sign bit and the payload of *x*, and is quiet if and only if *x* is quiet. Never raises a floating-point exception—not even for signalling NaN.

`Abs` is limited to real numbers. For the complex magnitude, use `magnitude` instead.

```
(abs -7)                ⇒  7
(abs 0.)                ⇒  0.
(abs -0.)               ⇒  0.
```

[4] MIT/GNU Scheme signals an error of type `condition-type:bad-range-argument` in this case.

```
(abs -inf.0)            ⇒  +inf.0
(abs -snan.123)         ⇒  +snan.123
(abs 1+2i)              error
```

`copysign` *x1 x2* [procedure]

Returns a real number with the magnitude of *x1* and the sign of *x2*. If *x1* is a NaN, then the result has the payload of *x1* and is quiet if and only if *x1* is quiet. Never raises a floating-point exception—not even for signalling NaN.

```
(copysign 123 -1)             ⇒  -123
(copysign 0. -1)              ⇒  -0.
(copysign -0. 0.)             ⇒  0.
(copysign -nan.123 0.)        ⇒  +nan.123
(copysign +snan.123 -0.)      ⇒  -snan.123
```

`euclidean/` *n1 n2* [procedure]
`floor/` *n1 n2* [procedure]
`ceiling/` *n1 n2* [procedure]
`truncate/` *n1 n2* [procedure]
`round/` *n1 n2* [procedure]

These procedures implement number-theoretic (integer) division. Given a numerator *n1* and denominator *n2*, they return two values *n3* and *n4*, respectively quotient (an integer) and remainder, such that

$$n_1 = n_2 n_3 + n_4, \qquad 0 \le |n_4| < |n_2|.$$

The procedures differ in choice of quotient and remainder:

- `euclidean/` chooses the remainder *n4* to be nonnegative, so that $0 \le n_4 < |n_2|$.

```
(euclidean/ 15 4)       ⇒  3, 3
(euclidean/ 13 4)       ⇒  3, 1
(euclidean/ 13 -4)      ⇒  -3, 1
(euclidean/ +1 4)       ⇒  0, 1
(euclidean/ 0 4)        ⇒  0, 0
(euclidean/ -1 4)       ⇒  -1, 3
(euclidean/ -13 4)      ⇒  -4, 3
(euclidean/ -13 -4)     ⇒  4, 3
```

The quotient part of euclidean division, which can be computed without the remainder using `euclidean-quotient`, is an odd function of the denominator:

```
(- (euclidean-quotient n1 (- n2)))
= (euclidean-quotient n1 n2)
```

The remainder part of euclidean division, which can be computed without the quotient using `euclidean-remainder`, is an even function of the denominator:

```
(euclidean-remainder n1 (- n2))
= (euclidean-remainder n1 n2)
```

When the denominator *n2* is a positive power of two, the quotient *n3* of euclidean division coincides with the result of an arithmetic shift:

```
(shift-right n1 s)
= (euclidean-quotient n1 (expt 2 s))
```

- `truncate/` chooses the quotient to have absolute value $|n_3| = \lfloor |n_1/n_2| \rfloor$; in other words, `truncate/` rounds the quotient *n3* toward zero.

```
(truncate/ 15 4)        ⇒  3, 3
(truncate/ 13 4)        ⇒  3, 1
(truncate/ 13 -4)       ⇒  -3, 1
(truncate/ +1 4)        ⇒  0, 1
(truncate/ 0 4)         ⇒  0, 0
(truncate/ -1 4)        ⇒  0, -1
(truncate/ -13 4)       ⇒  -3, -1
(truncate/ -13 -4)      ⇒  3, -1
```

  If the remainder *n4* is nonzero, then it is negative if and only if the numerator *n1* is negative. Like euclidean division, the quotient *n3* is an odd function of the denominator *n2* and the remainder *n4* is an even function of the denominator *n2*.

- `floor/` chooses the quotient *n3* to be $\lfloor n_1/n_2 \rfloor$; in other words, `floor/` rounds the quotient *n3* toward negative infinity.

```
(floor/ 15 4)           ⇒  3, 3
(floor/ 13 4)           ⇒  3, 1
(floor/ 13 -4)          ⇒  -4, -3
(floor/ +1 4)           ⇒  0, 1
(floor/ 0 4)            ⇒  0, 0
(floor/ -1 4)           ⇒  -1, 3
(floor/ -13 4)          ⇒  -4, 3
(floor/ -13 -4)         ⇒  3, -1
```

  If the remainder *n4* is nonzero, then it is negative if and only if the denominator *n2* is negative. Like euclidean division, when the denominator *n2* is a positive power of two, the quotient *n3* from `floor/` coincides with the result of an arithmetic shift.

- `ceiling/` chooses the quotient *n3* to be $\lceil n_1/n_2 \rceil$; in other words, `ceiling/` rounds the quotient *n3* toward positive infinity.

```
(ceiling/ 15 4)         ⇒  4, -1
(ceiling/ 13 4)         ⇒  4, -3
(ceiling/ 13 -4)        ⇒  -3, 1
(ceiling/ +1 4)         ⇒  1, -3
(ceiling/ 0 4)          ⇒  0, 0
(ceiling/ -1 4)         ⇒  0, -1
(ceiling/ -13 4)        ⇒  -3, -1
(ceiling/ -13 -4)       ⇒  4, 3
```

  If the remainder *n4* is nonzero, then it is negative if and only if the denominator *n2* is nonnegative. `Ceiling/` is useful for determining, given a number of units, how many blocks of a fixed number of units are needed to store that many units and how much space is wasted.

- `round/` chooses the quotient *n3* to be the integer nearest to n_1/n_2, or if it is exactly halfway between two integers, then the nearest even integer; in other

words, `round/` implements the default IEEE 754 rounding mode of round-to-nearest/ties-to-even.

```
(round/ 15 4)           ⇒  4, -1
(round/ 13 4)           ⇒  3, 1
(round/ 13 -4)          ⇒  -3, 1
(round/ +1 4)           ⇒  0, 1
(round/ 0 4)            ⇒  0, 0
(round/ -1 4)           ⇒  0, -1
(round/ -13 4)          ⇒  -3, -1
(round/ -13 -4)         ⇒  3, -1
```

`Round/` guarantees that $0 \leq |n_4| < |n_2|/2$. Like euclidean division, the quotient *n3* is an odd function of the denominator *n2* and the remainder *n4* is an even function of the denominator *n2*.

These procedures may also be applied to any real numbers, and are subject to inexact contagion:

```
(truncate/ -13 -4.0)    ⇒  3.0, -1.0   ; inexact
(floor/ 4.56 1.0)       ⇒  4.0, 0.56
(floor/ #e4.56 1)       ⇒  4, 14/25
(euclidean/ 7.853981633974483 (atan 0 -1))
                        ⇒  2.0, 1.5707963267948966
```

(These are not useful for precise floating-point argument reduction modulo 2π, e.g., in `cos`, `sin`, etc.; for that you need to represent π in extended precision.)

`euclidean-quotient` *n1 n2* [procedure]
`euclidean-remainder` *n1 n2* [procedure]
`floor-quotient` *n1 n2* [procedure]
`floor-remainder` *n1 n2* [procedure]
`ceiling-quotient` *n1 n2* [procedure]
`ceiling-remainder` *n1 n2* [procedure]
`truncate-quotient` *n1 n2* [procedure]
`truncate-remainder` *n1 n2* [procedure]
`round-quotient` *n1 n2* [procedure]
`round-remainder` *n1 n2* [procedure]

These procedures return only the quotient or remainder part of an integer division. For example, (`floor/` *n1 n2*) is equivalent to:

```
(values (floor-quotient n1 n2)
        (floor-remainder n1 n2))
```

`quotient` *n1 n2* [procedure]
`remainder` *n1 n2* [procedure]
`modulo` *n1 n2* [procedure]
`integer-floor` *n1 n2* [procedure]
`integer-ceiling` *n1 n2* [procedure]
`integer-truncate` *n1 n2* [procedure]
`integer-round` *n1 n2* [procedure]

Historic aliases for integer division routines, not applicable to non-integral rational or real numbers:

- `quotient` is an alias for `truncate-quotient`
- `remainder` is an alias for `truncate-remainder`
- `modulo` is an alias for `floor-remainder`
- `integer-floor` is an alias for `floor-quotient`
- `integer-ceiling` is an alias for `ceiling-quotient`
- `integer-truncate` is an alias for `truncate-quotient`
- `integer-round` is an alias for `round-quotient`

`integer-divide` *n1 n2* [procedure]
`integer-divide-quotient` *qr* [procedure]
`integer-divide-remainder` *qr* [procedure]

Historic alias for `truncate/` with a special data structure instead of multiple return values, not applicable to non-integral rational or real arguments. If *qr* is the result of `(integer-divide` *n1 n2*`)`, then `(truncate/` *n1 n2*`)` is equivalent to:

```
(values (integer-divide-quotient qr)
        (integer-divide-remainder qr))
```

`gcd` *n1 ...* [procedure]
`lcm` *n1 ...* [procedure]

These procedures return the greatest common divisor or least common multiple of their arguments. The result is always non-negative.

```
(gcd 32 -36)            ⇒  4
(gcd)                   ⇒  0

(lcm 32 -36)            ⇒  288
(lcm 32.0 -36)          ⇒  288.0  ; inexact
(lcm)                   ⇒  1
```

`modexp` *b e m* [procedure]

Modular exponentiation. Returns $b^e \bmod m$. *b*, *e*, and *m* must be exact integers; *m* must be nonzero. Mathematically equivalent to `(modulo (expt b e) m)`, but works even for large inputs for which the intermediate `(expt b e)` would overflow memory.

```
(modexp 1234 5678 90)   ⇒  46
(modexp 2 (expt 2 1024) 299)
                        ⇒  55
```

`numerator` *q* [procedure]
`denominator` *q* [procedure]

These procedures return the numerator or denominator of their argument; the result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

```
(numerator (/ 6 4))     ⇒  3
(denominator (/ 6 4))   ⇒  2
(denominator (inexact (/ 6 4)))
                        ⇒  2.0
```

`floor` *x* [procedure]
`ceiling` *x* [procedure]
`truncate` *x* [procedure]
`round` *x* [procedure]

These procedures return integers. `floor` returns the largest integer not larger than *x*. `ceiling` returns the smallest integer not smaller than *x*. `truncate` returns the integer closest to *x* whose absolute value is not larger than the absolute value of *x*. `round` returns the closest integer to *x*, rounding to even when *x* is halfway between two integers.

Rationale: `round` rounds to even for consistency with the rounding modes required by the IEEE floating point standard.

Note: If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result should be passed to the `exact` procedure (or use one of the procedures below).

```
(floor -4.3)          ⇒  -5.0
(ceiling -4.3)        ⇒  -4.0
(truncate -4.3)       ⇒  -4.0
(round -4.3)          ⇒  -4.0

(floor 3.5)           ⇒  3.0
(ceiling 3.5)         ⇒  4.0
(truncate 3.5)        ⇒  3.0
(round 3.5)           ⇒  4.0   ; inexact

(round 7/2)           ⇒  4     ; exact
(round 7)             ⇒  7
```

`floor->exact` *x* [procedure]
`ceiling->exact` *x* [procedure]
`truncate->exact` *x* [procedure]
`round->exact` *x* [procedure]

These procedures are similar to the preceding procedures except that they always return an exact result. For example, the following are equivalent

```
(floor->exact x)
(exact (floor x))
```

except that the former is faster and has fewer range restrictions.

`rationalize` *x y* [procedure]
`rationalize->exact` *x y* [procedure]

`rationalize` returns the *simplest* rational number differing from *x* by no more than *y*. A rational number *r1* is *simpler* than another rational number *r2* if *r1*`=`*p1*`/`*q1* and *r2*`=`*p2*`/`*q2* (both in lowest terms) and `|`*p1*`|<=|`*p2*`|` and `|`*q1*`|<=|`*q2*`|`. Thus `3/5` is simpler than `4/7`. Although not all rationals are comparable in this ordering (consider `2/7` and `3/5`) any interval contains a rational number that is simpler than every other rational number in that interval (the simpler `2/5` lies between `2/7` and `3/5`). Note that `0=0/1` is the simplest rational of all.

```
(rationalize (exact .3) 1/10)  ⇒  1/3    ; exact
(rationalize .3 1/10)          ⇒  #i1/3  ; inexact
```

`rationalize->exact` is similar to `rationalize` except that it always returns an exact result.

`simplest-rational` *x y* [procedure]
`simplest-exact-rational` *x y* [procedure]

`simplest-rational` returns the simplest rational number between *x* and *y* inclusive; `simplest-exact-rational` is similar except that it always returns an exact result.

These procedures implement the same functionality as `rationalize` and `rationalize->exact`, except that they specify the input range by its endpoints; `rationalize` specifies the range by its center point and its (half-) width.

`exp` *z* [procedure]
`log` *z* [procedure]
`sin` *z* [procedure]
`cos` *z* [procedure]
`tan` *z* [procedure]
`asin` *z* [procedure]
`acos` *z* [procedure]
`atan` *z* [procedure]
`atan` *y x* [procedure]

These procedures compute the usual transcendental functions. `log` computes the natural logarithm of *z* (not the base ten logarithm). `asin`, `acos`, and `atan` compute arcsine, arccosine, and arctangent, respectively. The two-argument variant of `atan` computes `(angle (make-rectangular x y))` (see below).

In general, the mathematical functions log, arcsine, arccosine, and arctangent are multiply defined. For nonzero real *x*, the value of log *x* is defined to be the one whose imaginary part lies in the range minus *pi* (exclusive) to *pi* (inclusive). log 0 is undefined. The value of log *z* when *z* is complex is defined according to the formula

$$\log z = \log \text{magnitude}(z) + i\,\text{angle}(z)$$

With log defined this way, the values of arcsine, arccosine, and arctangent are according to the following formulae:

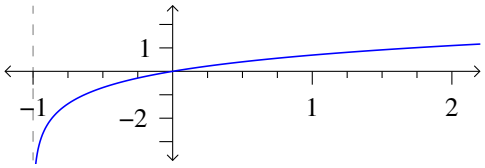
$$\sin^{-1} z = -i \log(iz + \sqrt{1 - z^2})$$
$$\cos^{-1} z = \pi/2 - \sin^{-1} z$$
$$\tan^{-1} z = (\log(1 + iz) - \log(1 - iz))/(2i)$$

The above specification follows *Common Lisp: the Language*, which in turn cites *Principal Values and Branch Cuts in Complex APL*; refer to these sources for more detailed discussion of branch cuts, boundary conditions, and implementation of these functions. When it is possible these procedures produce a real result from a real argument.

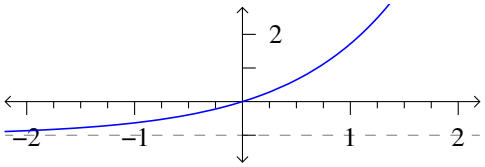
`log1p` *z* [procedure]
`expm1` *z* [procedure]

Equivalent to:

$$\operatorname{log1p} z = \log(1+z),$$



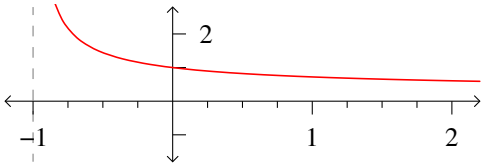
$$\operatorname{expm1} z = \exp(z) - 1.$$



However, for real numbers close to zero, or for complex numbers near the circle of radius 1 about −1, these provide better approximations than `(log (+ 1 z))` or `(- (exp z) 1)`:

- Floating-point numbers have much higher density around 0 than around 1, so naive translation from near 0 to near 1 loses precision, and naive computation of a number near 1 loses precision even if it is followed by translation to near 0.
- The condition number of log near 1 is unbounded, while the condition number of log1p near 0 is near 1:

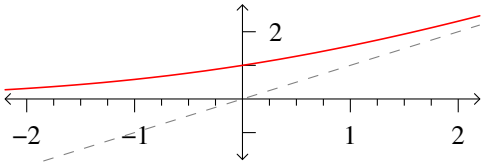
$$xf'(x)/f(x) = \frac{x/(1+x)}{\log(1+x)}.$$



(Conversely, the condition number of log near 0 approaches 0, while the condition number of log1p near −1 is unbounded, so for inputs near 0 it is better to compute them via log rather than via log1p.)

- Similarly, although the condition number of exp near 0 is near 0, its *value* near 0 is near 1, and the condition number of $y - 1$ is unbounded for y near 1, so the intermediate error introduced by `(exp z)` may be amplified arbitrarily by then computing `(- (exp z) 1)`. In contrast, the condition number of expm1 itself near 0 is near 1, so it does not inherently amplify errors:

$$xf'(x)/f(x) = \frac{xe^x}{e^x - 1}.$$



On real numbers, the forward relative error of this implementation is determined by the system's math library, usually below 1ulp.

On complex numbers:

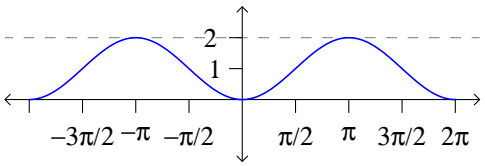
- The forward relative error *in complex magnitude* is bounded by a small multiple of the system math library's error for exp, expm1, log, log1p, sin, and cos.
- The *componentwise* forward relative error of `log1p` is bounded by a small multiple of the system math library's error for log and log1p.

- The *componentwise* forward relative error of `expm1` is bounded by a small multiple of the system math library's error for exp, expm1, log, log1p, sin, and cos—*except* on $x + iy$ where x is near zero and e^{-x} is near $\cos y$.

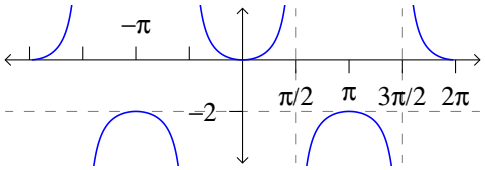
`versin` z [procedure]
`exsec` z [procedure]
`aversin` z [procedure]
`aexsec` z [procedure]

Zero-centered trigonometric functions related to cosine but well-conditioned near zero:

$$\operatorname{versin} z = 1 - \cos z,$$

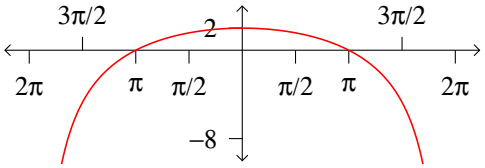


$$\operatorname{exsec} z = \frac{1 - \cos z}{\cos z}.$$

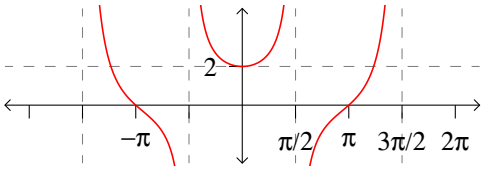


Both functions are well-conditioned near zero on the real line:

$$\frac{x \operatorname{versin}' x}{\operatorname{versin} x} = \frac{x \sin x}{1 - \cos x},$$



$$\frac{x \operatorname{exsec}' x}{\operatorname{exsec} x} = \frac{x \tan x}{1 - \cos x}.$$



`logp1` z [procedure]

Alias for `log1p`.

`exp2` z [procedure]
`exp10` z [procedure]
`exp2m1` z [procedure]
`exp10m1` z [procedure]
`log2` z [procedure]
`log10` z [procedure]
`log2p1` z [procedure]
`log10p1` z [procedure]

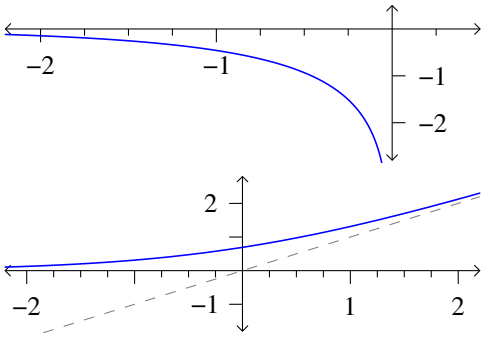
Base-2 and base-10 variants of `exp`, `expm1`, `log`, and `log1p`.

`log1mexp` x [procedure]
`log1pexp` x [procedure]

Equivalent to:

$$\operatorname{log1mexp} x = \log(1 - e^x),$$

$$\operatorname{log1pexp} x = \log(1 + e^x).$$



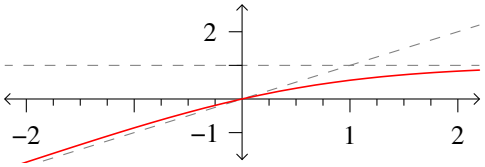
Like log1p and expm1, these avoid numerical pathologies with the intermediate quantities $1 - e^x$ and $1 + e^x$ and inputs to log near 1.

- log1mexp computes the complement of a probability p in log-space $\log p$, and as such is a self-inverse. It is finite when $x < 0$; negative infinity when $x = 0$; and invalid otherwise.
- log1pexp is related to the logistic function $1/(1 + e^{-x})$ — specifically, it differs from the logarithm of the logistic function only by the sign of the input and the output.

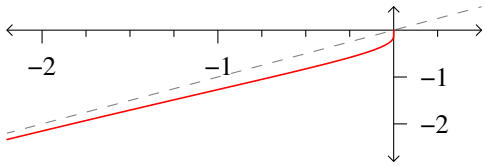
This implementation gives forward relative error bounded by ten times the forward relative error bound of the system math library's log and exp, which is usually below 1ulp.

Beware that although the forward relative error of the MIT/GNU Scheme *implementations* of these functions is bounded, these *functions* are ill-conditioned for large negative inputs:

$$xf'(x)/f(x) = \frac{\pm xe^x}{(1 \pm e^x)\log(1 \pm e^x)} \approx x, \quad \text{for } x \ll 0.$$



Condition number of log1pexp



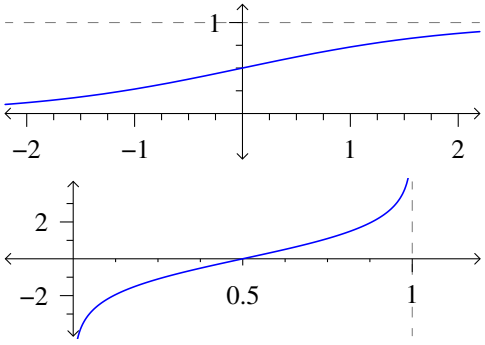
Condition number of log1mexp

`logistic` x [procedure]
`logit` x [procedure]

Logistic and logit functions. Equivalent to:

$$\operatorname{logistic} x = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}},$$

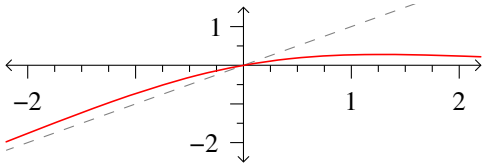
$$\operatorname{logit} p = \log \frac{p}{1 - p} = \log \frac{1}{\frac{1}{p} - 1}.$$



These functions are inverses of one another. The logit function maps a probablity p in $[0, 1]$ into log-odds x in the extended real line, and the logistic function maps back from log-odds to probabilities.

- The logistic function is defined on the entire real line, but is ill-conditioned for large negative x, with condition number

$$xf'(x)/f(x) = \frac{xe^{-x}}{1 + e^{-x}}.$$



  The identity

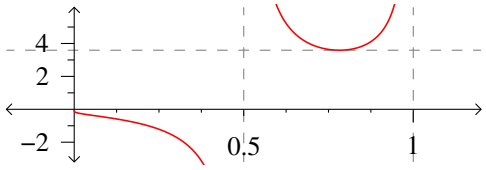
$$\text{logistic}(-x) = 1 - \text{logistic}(x)$$

  may help to rearrange a computation, along with the logistic-1/2 function which ranges from $-1/2$ to $+1/2$ and centered at zero rather than from 0 to 1 and centered at $1/2$.

  This implementation gives forward relative error bounded by at most seven times the forward relative error bound of the system math library's exp, which is usually below 1ulp.

- The logit function is defined on the closed unit interval $[0, 1]$, but is ill-conditioned near $1/2$ and 1, with condition number

$$xf'(x)/f(x) = \frac{1/(1-p)}{\log(p/(1-p))}.$$



  The identity

$$\text{logit}(1 - p) = -\text{logit}(p)$$

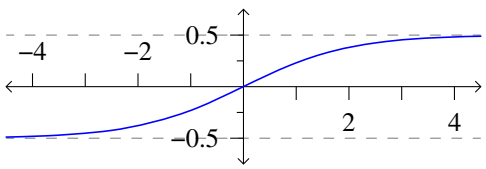
  may help to rearrange a computation, along with the logit1/2+ function which is defined on $-1/2$ to $+1/2$ and centered at zero rather than on 0 to 1 and centered at $1/2$.

  This implementation gives forward relative error bounded by at most ten times the forward relative error bound of the system math library's log, which is usually below 1ulp.

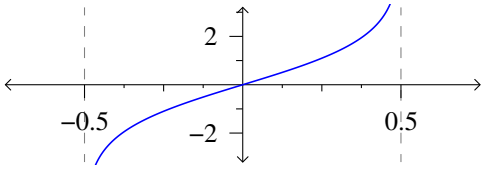
`logistic-1/2` *x* [procedure]
`logit1/2+` *x* [procedure]

Equivalent to:

$$\text{logistic-1/2}\, x = \text{logistic}(x) - 1/2,$$



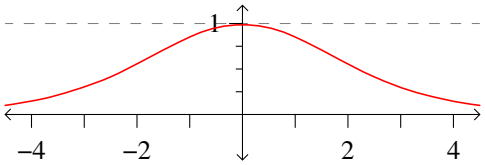
$$\text{logit1/2+}\, p = \text{logit}(1/2 + p).$$



Like `logistic` and `logit`, these functions are inverses of one another; unlike `logistic` and `logit`, their domains and codomains are both centered at zero.

- The logistic-1/2 function is well-conditioned on the entire real line, with maximum condition number 1 at 0:

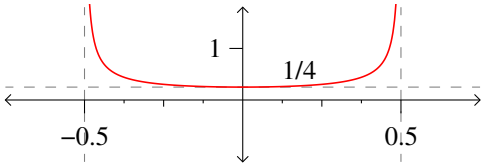
$$xf'(x)/f(x) = \frac{2xe^{-x}}{1 - e^{-2x}}.$$



  This implementation gives forward relative error bounded by 5 times the forward relative error bound of the system math library's exp, which is usually below 1ulp.

- The logit1/2+ function is defined on $[-1/2, +1/2]$, and is ill-conditioned near $-1/2$ and $+1/2$:

$$xf'(x)/f(x) = \frac{x/(1 - 4x^2)}{\text{logit}(1/2 + x)}.$$

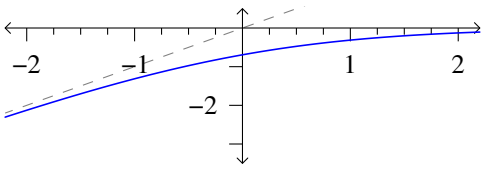


  For points near $-1/2$ or $+1/2$, it may be better to compute logit of a point near 0 instead. This implementation gives forward relative error bounded by 34 times the forward relative error bound of the system math library's log, which is usually below 1ulp.

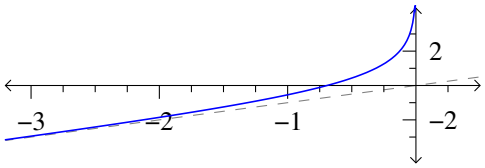
`log-logistic` *x* [procedure]
`logit-exp` *x* [procedure]

Equivalent to:

$$\begin{aligned}\text{log-logistic}\, x &= \log \text{logistic}(x) \\ &= \log[1/(1 + e^{-x})],\end{aligned}$$



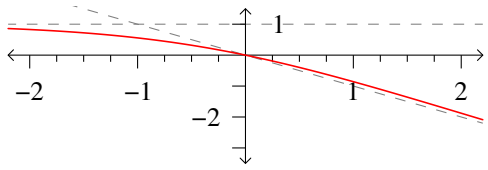
$$\begin{aligned}\text{logit-exp}\, x &= \text{logit}(e^x) \\ &= \log[e^x/(1 - e^x)].\end{aligned}$$



Like `logistic` and `logit`, these functions are inverses of one another.

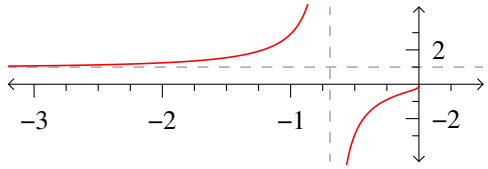
- The log-logistic function maps log-odds on the extended real line to log-probability on the nonpositive half of the extended real line, and is ill-conditioned for large positive x:

$$xf'(x)/f(x) = \frac{-xe^{-x}/(1+e^{-x})}{\log(1+e^{-x})}.$$



- The logit-exp function maps log-probability on the nonpositive half of the extended real line to log-odds on the extended real line, and is ill-conditioned near $\log(1/2)$:

$$xf'(x)/f(x) = \frac{x/(1-e^{x})}{\log[e^{x}/(1-e^{x})]}.$$



This implementation gives forward relative error bounded by ten times the forward relative error bound of the system math library's log and exp, which is usually below 1ulp.

`logsumexp` *list* [procedure]

List must be a list of real numbers $x_1, x_2, \ldots, x_n$. Returns an approximation to:

$$\log(e^{x_1} + e^{x_2} + \cdots + e^{x_n}).$$

The computation avoids intermediate overflow; `logsumexp` returns `+inf.0` if and only if one of the inputs is `+inf.0`.

`sin-pi*` x [procedure]
`cos-pi*` x [procedure]
`tan-pi*` x [procedure]
`versin-pi*` x [procedure]
`exsec-pi*` x [procedure]
`asin/pi` x [procedure]
`acos/pi` x [procedure]
`atan/pi` x [procedure]
`atan2/pi` y x [procedure]
`aversin/pi` x [procedure]
`aexsec/pi` x [procedure]

These procedures compute the standard trigonometric functions in units of half-revolutions rather than units of radians. Mathematically, `(sin-pi* x)` computes $\sin(\pi x)$ and `(asin/pi x)` computes $\arcsin(x)/\pi$, etc.

`sqrt` z [procedure]

Returns the principal square root of z. The result will have either positive real part, or zero real part and non-negative imaginary part.

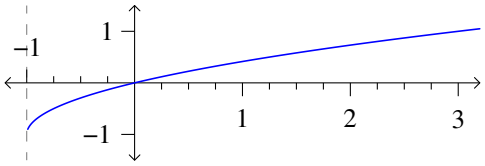
`rsqrt` z [procedure]

Returns the reciprocal of the principal square root of z.

`sqrt1pm1` z [procedure]

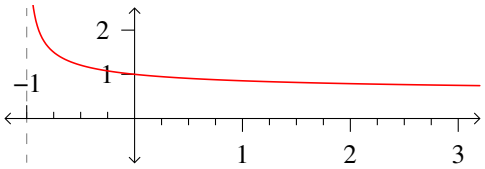
Returns

$$\sqrt{1+z}-1.$$



This function is well-conditioned except for z near -1; the condition number is:

$$\frac{z/2}{z-(\sqrt{1+z}-1)}.$$



Using `(sqrt1pm1 z)` instead of `(- (sqrt (+ 1 z)) 1)` avoids loss of precision when z is near 0.

`expt` *z1 z2* [procedure]

Returns *z1* raised to the power *z2*:

$$z_1{}^{z_2} = e^{z_2 \log z_1}$$

0^0 is defined to be equal to 1.

`compound` *z1 z2* [procedure]
`compoundm1` *z1 z2* [procedure]

`Compound` returns $(1+z_1)^{z_2}$, and `compoundm1` returns $(1+z_1)^{z_2}-1$, with low relative error even for *z1* near zero.

`make-rectangular` *x1 x2* [procedure]
`make-polar` *x3 x4* [procedure]
`real-part` *z* [procedure]
`imag-part` *z* [procedure]
`magnitude` *z* [procedure]
`angle` *z* [procedure]
`conjugate` *z* [procedure]

Suppose *x1*, *x2*, *x3*, and *x4* are real numbers and z is a complex number such that

$$z = x_1 + x_2 i = x_3 \cdot e^{i x_4}$$

Then `make-rectangular` and `make-polar` return z, `real-part` returns *x1*, `imag-part` returns *x2*, `magnitude` returns *x3*, and `angle` returns *x4*. In the case of `angle`, whose value is not uniquely determined by the preceding rule, the value returned will be the one in the range minus *pi* (exclusive) to *pi* (inclusive).

`conjugate` returns the complex conjugate of z.

The procedures `exact` and `inexact` implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

`inexact` *z* [procedure]
`exact->inexact` *z* [procedure]

`inexact` returns an inexact representation of z. The value returned is the inexact number that is numerically closest to the argument. For inexact arguments, the

result is the same as the argument. For exact complex numbers, the result is a complex number whose real and imaginary parts are the result of applying `inexact` to the real and imaginary parts of the argument, respectively. If an exact argument has no reasonably close inexact equivalent (in the sense of `=`), then a violation of an implementation restriction may be reported.

The procedure `exact->inexact` has been deprecated by R7RS.

`exact` *z* [procedure]
`inexact->exact` *z* [procedure]

`exact` returns an exact representation of *z*. The value returned is the exact number that is numerically closest to the argument. For exact arguments, the result is the same as the argument. For inexact non-integral real arguments, the implementation may return a rational approximation, or may report an implementation violation. For inexact complex arguments, the result is a complex number whose real and imaginary parts are the result of applying `exact` to the real and imaginary parts of the argument, respectively. If an inexact argument has no reasonably close exact equivalent (in the sense of `=`), then a violation of an implementation restriction may be reported.

The procedure `inexact->exact` has been deprecated by R7RS.

4.6 Numerical input and output

`number->string` *number* [*radix*] [procedure]

Radix must be an exact integer, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. The procedure `number->string` takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number radix)
                        radix)))
```

is true. It is an error if no possible result makes this expression true.

If *number* is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true; otherwise the format of the result is unspecified.

The result returned by `number->string` never contains an explicit radix prefix.

Note: The error case can occur only when *number* is not a complex number or is a complex number with an non-rational real or imaginary part.

Rationale: If *number* is an inexact number represented using flonums, and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and non-flonum representations.

`flonum-parser-fast?` [variable]

This variable controls the behavior of `string->number` when parsing inexact numbers. Specifically, it allows the user to trade off accuracy against speed.

When set to its default value, `#f`, the parser provides maximal accuracy, as required by the Scheme standard. If set to `#t`, the parser uses faster algorithms that will sometimes introduce small errors in the result. The errors affect a few of the least-significant bits of the result, and consequently can be tolerated by many applications.

`flonum-unparser-cutoff` [variable]

This variable is **deprecated**; use `param:flonum-printer-cutoff` instead.

`param:flonum-printer-cutoff` [parameter]

This parameter controls the action of `number->string` when *number* is a flonum (and consequently controls all printing of flonums). This parameter may be called with an argument to set its value.

The value of this parameter is normally a list of three items:

rounding-type
: One of the following symbols: `normal`, `relative`, or `absolute`. The symbol `normal` means that the number should be printed with full precision. The symbol `relative` means that the number should be rounded to a specific number of digits. The symbol `absolute` means that the number should be rounded so that there are a specific number of digits to the right of the decimal point.

precision
: An exact integer. If *rounding-type* is `normal`, *precision* is ignored. If *rounding-type* is `relative`, *precision* must be positive, and it specifies the number of digits to which the printed representation will be rounded. If *rounding-type* is `absolute`, the printed representation will be rounded *precision* digits to the right of the decimal point; if *precision* is negative, the representation is rounded `(-` *precision*`)` digits to the left of the decimal point.

format-type
: One of the symbols: `normal`, `scientific`, or `engineering`. This specifies the format in which the number will be printed.
`scientific` specifies that the number will be printed using scientific notation: *x*`.`*xxx*`e`*yyy*. In other words, the number is printed as a significand between zero inclusive and ten exclusive, and an exponent. `engineering` is like `scientific`, except that the exponent is always a power of three, and the significand is constrained to be between zero inclusive and 1000 exclusive. If `normal` is specified, the number will be printed in positional notation if it is "small enough", otherwise it is printed in scientific notation. A number is "small enough" when the number of digits that would be printed using positional notation does not exceed the number of digits of precision in the underlying floating-point number representation; IEEE 754-2008 binary64 floating-point numbers have 17 digits of precision.

This three-element list may be abbreviated in two ways. First, the symbol `normal` may be used, which is equivalent to the list `(normal 0 normal)`. Second, the third element of the list, *format-type*, may be omitted, in which case it defaults to `normal`.

The default value for `param:flonum-printer-cutoff` is `normal`. If it is bound to a value different from those described here, `number->string` issues a warning and acts as though the value had been `normal`.

Some examples of `param:flonum-printer-cutoff`:

```
(number->string (* 4 (atan 1 1)))
                                        ⇒  "3.141592653589793"
(parameterize ((param:flonum-printer-cutoff '(relative 5)))
  (number->string (* 4 (atan 1 1))))
                                        ⇒  "3.1416"
(parameterize ((param:flonum-printer-cutoff '(relative 5)))
  (number->string (* 4000 (atan 1 1))))
                                        ⇒  "3141.6"
(parameterize ((param:flonum-printer-cutoff '(relative 5 scientific)))
  (number->string (* 4000 (atan 1 1))))
                                        ⇒  "3.1416e3"
(parameterize ((param:flonum-printer-cutoff '(relative 5 scientific)))
  (number->string (* 40000 (atan 1 1))))
                                        ⇒  "3.1416e4"
(parameterize ((param:flonum-printer-cutoff '(relative 5 engineering)))
  (number->string (* 40000 (atan 1 1))))
                                        ⇒  "31.416e3"
(parameterize ((param:flonum-printer-cutoff '(absolute 5)))
  (number->string (* 4 (atan 1 1))))
                                        ⇒  "3.14159"
(parameterize ((param:flonum-printer-cutoff '(absolute 5)))
  (number->string (* 4000 (atan 1 1))))
                                        ⇒  "3141.59265"
(parameterize ((param:flonum-printer-cutoff '(absolute -4)))
  (number->string (* 4e10 (atan 1 1))))
                                        ⇒  "31415930000."
(parameterize ((param:flonum-printer-cutoff '(absolute -4 scientific)))
  (number->string (* 4e10 (atan 1 1))))
                                        ⇒  "3.141593e10"
(parameterize ((param:flonum-printer-cutoff '(absolute -4 engineering)))
  (number->string (* 4e10 (atan 1 1))))
                                        ⇒  "31.41593e9"
(parameterize ((param:flonum-printer-cutoff '(absolute -5)))
  (number->string (* 4e10 (atan 1 1))))
                                        ⇒  "31415900000."
```

`string->number` *string* [*radix*] [procedure]

Returns 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`.

```
(string->number "100")        ⇒  100
(string->number "100" 16)     ⇒  256
(string->number "1e2")        ⇒  100.0
(string->number "15##")       ⇒  1500.0
```

Note that a numeric representation using a decimal point or an exponent marker is not recognized unless *radix* is `10`.

4.7 Bit operations

This section describes operations on exact integers as strings of bits in two's-complement representation. All arguments must be exact integers.

`bitwise-not` *x* [procedure]

Bitwise complement. Equivalent to `(- -1 x)`.

`bitwise-and` *x1 x2 ...* [procedure]
`bitwise-andc2` *x1 x2* [procedure]
`bitwise-andc1` *x1 x2* [procedure]
`bitwise-xor` *x1 x2 ...* [procedure]
`bitwise-ior` *x1 x2 ...* [procedure]
`bitwise-nor` *x1 x2* [procedure]
`bitwise-eqv` *x1 x2 ...* [procedure]
`bitwise-orc2` *x1 x2* [procedure]
`bitwise-orc1` *x1 x2* [procedure]
`bitwise-nand` *x1 x2* [procedure]

Bitwise operations. The 'c1'/'c2' variants take the complement of their first or second operand, respectively; for example, `(bitwise-andc2 x1 x2)` is equivalent to `(bitwise-and x1 (bitwise-not x2))`.

The four bitwise operations that are associative are extended to arbitrary numbers of arguments with an appropriate identity:

```
(bitwise-and)          ⇒ -1
(bitwise-xor)          ⇒ 0
(bitwise-ior)          ⇒ 0
(bitwise-eqv)          ⇒ -1
```

`shift-left` *x n* [procedure]
`shift-right` *x n* [procedure]
`arithmetic-shift` *x n* [procedure]

Multiplication and integer division by 2^n: `(shift-left x n)` is equivalent to `(* x (expt 2 n))`, and `(shift-left x n)` is equivalent to `(euclidean-quotient x (expt 2 n))`.

`Shift-left` and `shift-right` require *n* to be nonnegative; `arithmetic-shift` is equivalent to `shift-left` for positive *n*, and equivalent to `shift-right` for the negation of negative *n*.

`bit-mask` *size position* [procedure]

Returns an integer with *size* consecutive bits set starting at *position*, zero-based, and all other bits clear. Both arguments must be nonnegative.

```
(bit-mask 0 123)        ⇒ 0
(bit-mask 4 3)          ⇒ #b1111000
```

`bit-antimask` *size position* [procedure]

Returns an integer with *size* consecutive bits clear starting at *position*, zero-based, and all other bits set. Both arguments must be nonnegative. Equivalent to `(bitwise-not (bit-mask` *size position*`))`.

```
(bit-antimask 0 123)    ⇒ -1
(bit-antimask 4 3)      ⇒ #b-1111001  ; '#b...10000111'
```

`bit-count` *x* [procedure]

Returns the number of 1 bits in *x*, if it is nonnegative, or the number of 0 bits, if it is negative. Sometimes known as 'pop count' or 'population count'.

`hamming-distance` *x y* [procedure]

Returns the Hamming distance between *x* and *y*—that is, the number of bits that differ in corresponding positions of their finite representations, or −1 if they have opposite signs (meaning that they differ in an infinite number of positions). Equivalent to `(bit-count (bitwise-xor` *x y*`))`.

```
(hamming-distance 1 3)          ⇒ 1
(hamming-distance 7 8)          ⇒ 4
(hamming-distance -8 -9)        ⇒ 4
(hamming-distance 1 -1)         ⇒ -1
```

`integer-length` *x* [procedure]

Returns the number of bit positions needed to represent *x* in two's-complement: zero when *x* is zero; the exact value of `(ceiling (/ (log x) (log 2)))` when *x* is positive; `(integer-length (bitwise-not x))` when *x* is negative.

```
(integer-length -129)           ⇒ 9
(integer-length -128)           ⇒ 8
(integer-length -127)           ⇒ 7
(integer-length -1)             ⇒ 0
(integer-length 0)              ⇒ 0
(integer-length 1)              ⇒ 1
(integer-length 127)            ⇒ 7
(integer-length 128)            ⇒ 8
```

`first-set-bit` *x* [procedure]

Returns the zero-based position of the first set bit in *x*. For zero, returns −1.

`set-bit` *n x* [procedure]
`clear-bit` *n x* [procedure]
`toggle-bit` *n x* [procedure]
`extract-bit` *n x* [procedure]
`bit-set?` *n x* [procedure]
`bit-clear?` *n x* [procedure]

`Set-bit` returns the integer *x* with the bit at position *n* set. `Clear-bit` returns the integer *x* with the bit at position *n* clear. `Toggle-bit` returns the integer *x* with the bit at position *n* in the opposite state.

`Extract-bit` returns the bit at position *n* in *x* as an integer, 0 or 1. `Bit-set?` returns true if that bit is set, false if clear. `Bit-clear?` returns false if that bit is set, true if clear.

`bit` *n* [procedure]
`bits` *n m* [procedure]

`(bit n)` returns a mask with the *n*th bit set and all other bits clear. `(bits n m)` returns a mask with bits *n* through *m* set, inclusive and zero-based, and all other bits clear. *n* and *m* may occur in either order.

```
(bit 7)                 ⇒ 128
(bits 4 5)              ⇒ #b110000
(bits 5 4)              ⇒ #b110000
```

`shiftout` *x mask* [procedure]
`shiftin` *x mask* [procedure]

mask must be a nonnegative integer with a contiguous substring of bits set, representing a contiguous field of bits.

`(shiftout x mask)` returns the corresponding value of that field in *x*; that is, `(bitwise-and (shift-right x n) mask)`, where *n* is the position of the first set bit in *mask*.

`(shiftin x mask)` returns an an integer with that field set to *x*; that is, `(shift-left x n)`.

Intended to be used with `bit` and `bits`. For example:

```
(define foo:x (bits 0 3))
(define foo:y (bits 4 5))
(define foo:z (bits 6 7))

(define (make-foo x y z)
  (bitwise-ior (shiftin x foo:x)
               (shiftin y foo:y)
               (shiftin z foo:z)))

(define (foo-x foo) (shiftout foo foo:x))
(define (foo-y foo) (shiftout foo foo:y))
(define (foo-z foo) (shiftout foo foo:z))

(make-foo 1 2 3)                ⇒ #b11100001
(foo-z (make-foo 1 2 3))        ⇒ 3
```

4.8 Fixnum and Flonum Operations

This section describes numerical operations that are restricted forms of the operations described above. These operations are useful because they compile very efficiently. However, care should be exercised: if used improperly, these operations can return incorrect answers, or even malformed objects that confuse the garbage collector.

4.8.1 Fixnum Operations

A *fixnum* is an exact integer that is small enough to fit in a machine word. In MIT/GNU Scheme, fixnums are typically 24 or 26 bits, depending on the machine; it is reasonable to assume that fixnums are at least 24 bits. Fixnums are signed; they are encoded using 2's complement.

All exact integers that are small enough to be encoded as fixnums are always encoded as fixnums — in other words, any exact integer that is not a fixnum is too big to be encoded as such. For this reason, small constants such as `0` or `1` are guaranteed to be fixnums.

`fix:fixnum?` *object* [procedure]
Returns `#t` if *object* is a fixnum; otherwise returns `#f`.

Here is an expression that determines the largest fixnum:

```
(let loop ((n 1))
  (if (fix:fixnum? n)
      (loop (* n 2))
      (- n 1)))
```

A similar expression determines the smallest fixnum.

`fix:=` *fixnum fixnum* [procedure]
`fix:<` *fixnum fixnum* [procedure]
`fix:>` *fixnum fixnum* [procedure]
`fix:<=` *fixnum fixnum* [procedure]
`fix:>=` *fixnum fixnum* [procedure]
These are the standard order and equality predicates on fixnums. When compiled, they do not check the types of their arguments.

`fix:zero?` *fixnum* [procedure]
`fix:positive?` *fixnum* [procedure]
`fix:negative?` *fixnum* [procedure]
These procedures compare their argument to zero. When compiled, they do not check the type of their argument. The code produced by the following expressions is identical:

```
(fix:zero? fixnum)
(fix:= fixnum 0)
```

Similarly, `fix:positive?` and `fix:negative?` produce code identical to equivalent expressions using `fix:>` and `fix:<`.

`fix:+` *fixnum fixnum* [procedure]
`fix:-` *fixnum fixnum* [procedure]
`fix:*` *fixnum fixnum* [procedure]
`fix:quotient` *fixnum fixnum* [procedure]
`fix:remainder` *fixnum fixnum* [procedure]
`fix:gcd` *fixnum fixnum* [procedure]
`fix:1+` *fixnum* [procedure]
`fix:-1+` *fixnum* [procedure]
These procedures are the standard arithmetic operations on fixnums. When compiled, they do not check the types of their arguments. Furthermore, they do not check to

see if the result can be encoded as a fixnum. If the result is too large to be encoded as a fixnum, a malformed object is returned, with potentially disastrous effect on the garbage collector.

`fix:divide` *fixnum fixnum* [procedure]
This procedure is like `integer-divide`, except that its arguments and its results must be fixnums. It should be used in conjunction with `integer-divide-quotient` and `integer-divide-remainder`.

The following are *bitwise-logical* operations on fixnums.

`fix:not` *fixnum* [procedure]
This returns the bitwise-logical inverse of its argument. When compiled, it does not check the type of its argument.

```
(fix:not 0)             ⇒  -1
(fix:not -1)            ⇒  0
(fix:not 1)             ⇒  -2
(fix:not -34)           ⇒  33
```

`fix:and` *fixnum fixnum* [procedure]
This returns the bitwise-logical "and" of its arguments. When compiled, it does not check the types of its arguments.

```
(fix:and #x43 #x0f)     ⇒  3
(fix:and #x43 #xf0)     ⇒  #x40
```

`fix:andc` *fixnum fixnum* [procedure]
Returns the bitwise-logical "and" of the first argument with the bitwise-logical inverse of the second argument. When compiled, it does not check the types of its arguments.

```
(fix:andc #x43 #x0f)    ⇒  #x40
(fix:andc #x43 #xf0)    ⇒  3
```

`fix:or` *fixnum fixnum* [procedure]
This returns the bitwise-logical "inclusive or" of its arguments. When compiled, it does not check the types of its arguments.

```
(fix:or #x40 3)         ⇒  #x43
(fix:or #x41 3)         ⇒  #x43
```

`fix:xor` *fixnum fixnum* [procedure]
This returns the bitwise-logical "exclusive or" of its arguments. When compiled, it does not check the types of its arguments.

```
(fix:xor #x40 3)        ⇒  #x43
(fix:xor #x41 3)        ⇒  #x42
```

`fix:lsh` *fixnum1 fixnum2* [procedure]
This procedure returns the result of shifting *fixnum1* by *fixnum2* bits. If *fixnum2* is positive, *fixnum1* is shifted left; if negative, it is shifted right *arithmetically* in signed

two's-complement. When compiled, it does not check the types of its arguments, nor the validity of its result.

```
(fix:lsh 1 10)          ⇒  #x400
(fix:lsh #x432 -10)     ⇒  1
(fix:lsh -1 3)          ⇒  -8
(fix:lsh -128 -4)       ⇒  -8
```

Warning: In the past, the semantics of `fix:lsh` when both arguments were negative has been inconsistent.

4.8.2 Flonum Operations

A *flonum* is an inexact real number that is implemented as a floating-point number. In MIT/GNU Scheme, all inexact real numbers are flonums. For this reason, constants such as `0.` and `2.3` are guaranteed to be flonums.

MIT/GNU Scheme follows the IEEE 754-2008 floating-point standard, using binary64 arithmetic for flonums. All floating-point values are classified into:

normal
: Numbers of the form
$$\pm r^e(1 + f/r^p)$$
where r, the radix, is a positive integer, here always 2; p, the precision, is a positive integer, here always 53; e, the exponent, is an integer within a limited range, here always -1022 to 1023 (inclusive); and f, the fractional part of the significand, is a $(p-1)$-bit unsigned integer,

subnormal
: Fixed-point numbers near zero that allow for gradual underflow. Every subnormal number is an integer multiple of the smallest subnormal number. Subnormals were also historically called "denormal".

zero
: There are two distinguished zero values, one with "negative" sign bit and one with "positive" sign bit.

  The two zero values are considered numerically equal, but serve to distinguish paths converging to zero along different branch cuts and so some operations yield different results for differently signed zero values.

infinity
: There are two distinguished infinity values, negative infinity or `-inf.0` and positive infinity or `+inf.0`, representing overflow on the real line.

NaN
: There are $4r^{p-2} - 2$ distinguished not-a-number values, representing invalid operations or uninitialized data, distinguished by their negative/positive sign bit, a quiet/signalling bit, and a $(p-2)$-digit unsigned integer payload which must not be zero for signalling NaNs.

  Arithmetic on **quiet** NaNs propagates them without raising any floating-point exceptions. In contrast, arithmetic on **signalling** NaNs raises the floating-point invalid-operation exception. Quiet NaNs are written `+nan.123`, `-nan.0`, etc. Signalling NaNs are written `+snan.123`, `-snan.1`, etc. The notation `+snan.0` and `-snan.0` is not allowed: what would be the encoding for them actually means `+inf.0` and `-inf.0`.

`flo:flonum?` *object* [procedure]
: Returns `#t` if *object* is a flonum; otherwise returns `#f`.

`flo:=` *flonum1 flonum2* [procedure]
`flo:<` *flonum1 flonum2* [procedure]
`flo:<=` *flonum1 flonum2* [procedure]
`flo:>` *flonum1 flonum2* [procedure]
`flo:>=` *flonum1 flonum2* [procedure]
`flo:<>` *flonum1 flonum2* [procedure]

These procedures are the standard order and equality predicates on flonums. When compiled, they do not check the types of their arguments. These predicates raise floating-point invalid-operation exceptions on NaN arguments; in other words, they are "ordered comparisons". When floating-point exception traps are disabled, they return false when any argument is NaN.

Every pair of floating-point numbers — excluding NaN — exhibits ordered trichotomy: they are related either by `flo:=`, `flo:<`, or `flo:>`.

`flo:safe=` *flonum1 flonum2* [procedure]
`flo:safe<` *flonum1 flonum2* [procedure]
`flo:safe<=` *flonum1 flonum2* [procedure]
`flo:safe>` *flonum1 flonum2* [procedure]
`flo:safe>=` *flonum1 flonum2* [procedure]
`flo:safe<>` *flonum1 flonum2* [procedure]
`flo:unordered?` *flonum1 flonum2* [procedure]

These procedures are the standard order and equality predicates on flonums. When compiled, they do not check the types of their arguments. These predicates do not raise floating-point exceptions, and simply return false on NaN arguments, except `flo:unordered?` which returns true iff at least one argument is NaN; in other words, they are "unordered comparisons".

Every pair of floating-point values — including NaN — exhibits unordered tetrachotomy: they are related either by `flo:safe=`, `flo:safe<`, `flo:safe>`, or `flo:unordered?`.

`flo:zero?` *flonum* [procedure]
`flo:positive?` *flonum* [procedure]
`flo:negative?` *flonum* [procedure]

Each of these procedures compares its argument to zero. When compiled, they do not check the type of their argument. These predicates raise floating-point invalid-operation exceptions on NaN arguments; in other words, they are "ordered comparisons".

```
(flo:zero? -0.)             ⇒  #t
(flo:negative? -0.)         ⇒  #f
(flo:negative? -1.)         ⇒  #t

(flo:zero? 0.)              ⇒  #t
(flo:positive? 0.)          ⇒  #f
(flo:positive? 1.)          ⇒  #f

(flo:zero? +nan.123)        ⇒  #f   ; (raises invalid-operation)
```

`flo:normal?` *flonum* [procedure]
`flo:subnormal?` *flonum* [procedure]
`flo:safe-zero?` *flonum* [procedure]
`flo:infinite?` *flonum* [procedure]
`flo:nan?` *flonum* [procedure]

Floating-point classification predicates. For any flonum, exactly one of these predicates returns true. These predicates never raise floating-point exceptions.

```
(flo:normal? 1.23)              ⇒  #t
(flo:subnormal? 4e-124)         ⇒  #t
(flo:safe-zero? -0.)            ⇒  #t
(flo:infinite? +inf.0)          ⇒  #t
(flo:nan? -nan.123)             ⇒  #t
```

`flo:finite?` *flonum* [procedure]

Equivalent to:

```
(or (flo:safe-zero? flonum)
    (flo:subnormal? flonum)
    (flo:normal? flonum))
; or
(and (not (flo:infinite? flonum))
     (not (flo:nan? flonum)))
```

True for normal, subnormal, and zero floating-point values; false for infinity and NaN.

`flo:classify` *flonum* [procedure]

Returns a symbol representing the classification of the flonum, one of `normal`, `subnormal`, `zero`, `infinity`, or `nan`.

`flo:sign-negative?` *flonum* [procedure]

Returns true if the sign bit of *flonum* is negative, and false otherwise. Never raises a floating-point exception—not even for signalling NaN.

```
(flo:sign-negative? +0.)        ⇒  #f
(flo:sign-negative? -0.)        ⇒  #t
(flo:sign-negative? -1.)        ⇒  #t
(flo:sign-negative? +inf.0)     ⇒  #f
(flo:sign-negative? +nan.123)   ⇒  #f

(flo:negative? -0.)             ⇒  #f
(flo:negative? +nan.123)        ⇒  #f   ; (raises invalid-operation)
```

`flo:+` *flonum1 flonum2* [procedure]
`flo:-` *flonum1 flonum2* [procedure]
`flo:*` *flonum1 flonum2* [procedure]
`flo:/` *flonum1 flonum2* [procedure]

These procedures are the standard arithmetic operations on flonums. When compiled, they do not check the types of their arguments.

`flo:*+` *flonum1 flonum2 flonum3* [procedure]
`flo:*-` *flonum1 flonum2 flonum3* [procedure]

`flo:fma` *flonum1 flonum2 flonum3* [procedure]
`flo:fast-fma?` [procedure]

Fused multiply-add: `(flo:*+ u v a)` computes $uv + a$ correctly rounded, with no intermediate overflow or underflow arising from uv. In contrast, `(flo:+ (flo:* u v) a)` may have two rounding errors, and can overflow or underflow if uv is too large or too small even if $uv + a$ is normal. `Flo:fma` is an alias for `flo:*+` with the more familiar name used in other languages like C.

`(flo:*- u v s)` computes $uv - s$ correctly rounded, equivalent to `(flo:*+ u v (flo:negate s))`.

`Flo:fast-fma?` returns true if the implementation of fused multiply-add is supported by fast hardware, and false if it is emulated using Dekker's double-precision algorithm in software.

```
(flo:+ (flo:* 1.2e100 2e208) -1.4e308)
                                  ⇒  +inf.0  ; (raises overflow)
(flo:*+ 1.2e100 2e208  -1.4e308)
                                  ⇒  1e308
```

`flo:negate` *flonum* [procedure]

This procedure returns the negation of its argument. When compiled, it does not check the type of its argument. Never raises a floating-point exception—not even for signalling NaN.

This is *not* equivalent to `(flo:- 0. flonum)`:

```
(flo:negate 1.2)              ⇒  -1.2
(flo:negate -nan.123)         ⇒  +nan.123
(flo:negate +inf.0)           ⇒  -inf.0
(flo:negate 0.)               ⇒  -0.
(flo:negate -0.)              ⇒  0.

(flo:- 0. 1.2)                ⇒  -1.2
(flo:- 0. -nan.123)           ⇒  -nan.123
(flo:- 0. +inf.0)             ⇒  -inf.0
(flo:- 0. 0.)                 ⇒  0.
(flo:- 0. -0.)                ⇒  0.
```

`flo:abs` *flonum* [procedure]
`flo:copysign` *flonum1 flonum2* [procedure]
`flo:exp` *flonum* [procedure]
`flo:exp2` *flonum* [procedure]
`flo:exp10` *flonum* [procedure]
`flo:expm1` *flonum* [procedure]
`flo:exp2m1` *flonum* [procedure]
`flo:exp10m1` *flonum* [procedure]
`flo:log` *flonum* [procedure]
`flo:log2` *flonum* [procedure]
`flo:log10` *flonum* [procedure]
`flo:log1p` *flonum* [procedure]

`flo:logp1` *flonum* [procedure]
`flo:log2p1` *flonum* [procedure]
`flo:log10p1` *flonum* [procedure]
`flo:sin` *flonum* [procedure]
`flo:cos` *flonum* [procedure]
`flo:tan` *flonum* [procedure]
`flo:asin` *flonum* [procedure]
`flo:acos` *flonum* [procedure]
`flo:atan` *flonum* [procedure]
`flo:sin-pi*` *flonum* [procedure]
`flo:cos-pi*` *flonum* [procedure]
`flo:tan-pi*` *flonum* [procedure]
`flo:asin/pi` *flonum* [procedure]
`flo:acos/pi` *flonum* [procedure]
`flo:atan/pi` *flonum* [procedure]
`flo:versin` *flonum* [procedure]
`flo:exsec` *flonum* [procedure]
`flo:aversin` *flonum* [procedure]
`flo:aexsec` *flonum* [procedure]
`flo:versin-pi*` *flonum* [procedure]
`flo:exsec-pi*` *flonum* [procedure]
`flo:aversin/pi` *flonum* [procedure]
`flo:aexsec/pi` *flonum* [procedure]
`flo:sinh` *flonum* [procedure]
`flo:cosh` *flonum* [procedure]
`flo:tanh` *flonum* [procedure]
`flo:asinh` *flonum* [procedure]
`flo:acosh` *flonum* [procedure]
`flo:atanh` *flonum* [procedure]
`flo:sqrt` *flonum* [procedure]
`flo:cbrt` *flonum* [procedure]
`flo:rsqrt` *flonum* [procedure]
`flo:sqrt1pm1` *flonum* [procedure]
`flo:expt` *flonum1 flonum2* [procedure]
`flo:compound` *flonum1 flonum2* [procedure]
`flo:compoundm1` *flonum1 flonum2* [procedure]
`flo:erf` *flonum* [procedure]
`flo:erfc` *flonum* [procedure]
`flo:hypot` *flonum1 flonum2* [procedure]
`flo:j0` *flonum* [procedure]
`flo:j1` *flonum* [procedure]
`flo:jn` *flonum* [procedure]
`flo:y0` *flonum* [procedure]
`flo:y1` *flonum* [procedure]
`flo:yn` *flonum* [procedure]
`flo:gamma` *flonum* [procedure]
`flo:lgamma` *flonum* [procedure]

`flo:floor` *flonum* [procedure]
`flo:ceiling` *flonum* [procedure]
`flo:truncate` *flonum* [procedure]
`flo:round` *flonum* [procedure]
`flo:floor->exact` *flonum* [procedure]
`flo:ceiling->exact` *flonum* [procedure]
`flo:truncate->exact` *flonum* [procedure]
`flo:round->exact` *flonum* [procedure]

These procedures are flonum versions of the corresponding procedures. When compiled, they do not check the types of their arguments.

`flo:atan2` *flonum1 flonum2* [procedure]
`flo:atan2/pi` *flonum1 flonum2* [procedure]

These are the flonum versions of `atan` and `atan/pi` with two arguments. When compiled, they do not check the types of their arguments.

`flo:signed-lgamma` *x* [procedure]

Returns two values,

$$m = \log|\Gamma(x)| \quad \text{and} \quad s = \operatorname{sign}\Gamma(x),$$

respectively a flonum and an exact integer either `-1` or `1`, so that

$$\Gamma(x) = s \cdot e^m.$$

`flo:min` *x1 x2* [procedure]
`flo:max` *x1 x2* [procedure]

Returns the min or max of two floating-point numbers. `-0.` is considered less than `+0.` for the purposes of `flo:min` and `flo:max`.

If either argument is NaN, raises the floating-point invalid-operation exception if it is a signalling NaN, and returns a quiet NaN. In other words, `flo:min` and `flo:max` propagate NaN.

These are the minimum and maximum operations of IEEE 754-2019.

`flo:min-mag` *x1 x2* [procedure]
`flo:max-mag` *x1 x2* [procedure]

Returns the argument that has the smallest or largest magnitude, or the min or max if the magnitude is the same.

If either argument is NaN, raises the floating-point invalid-operation exception if it is a signalling NaN, and returns a quiet NaN. In other words, `flo:min-mag` and `flo:max-mag` propagate NaN.

These are the minimumMagnitude and maximumMagnitude operations of IEEE 754-2019.

`flo:min-num` *x1 x2* [procedure]
`flo:max-num` *x1 x2* [procedure]

Returns the min or max of two floating-point numbers. `-0.` is considered less than `+0.` for the purposes of `flo:min-num` and `flo:max-num`.

If either argument is NaN, raises the floating-point invalid-operation exception if it is a signalling NaN, and returns the other one if it is not NaN, or the first argument if they are both NaN. In other words, `flo:min-num` and `flo:max-num` treat NaN as missing data and ignore it if possible.

These are the minimumNumber and maximumNumber operations of IEEE 754-2019, formerly called minNum and maxNum in IEEE 754-2008.

`flo:min-mag-num` *x1 x2* [procedure]
`flo:max-mag-num` *x1 x2* [procedure]

Returns the argument that has the smallest or largest magnitude, or the min or max if the magnitude is the same.

If either argument is NaN, raises the floating-point invalid-operation exception if it is a signalling NaN, and returns the other one if it is not NaN, or the first argument if they are both NaN. In other words, `flo:min-mag-num` and `flo:max-mag-num` treat NaN as missing data and ignore it if possible.

These are the minimumMagnitudeNumber and maximumMagnitudeNumber operations of IEEE 754-2019, formerly called minNumMag and maxNumMag in IEEE 754-2008.

`flo:ldexp` *x1 x2* [procedure]
`flo:scalbn` *x1 x2* [procedure]

`Flo:ldexp` scales by a power of two; `flo:scalbn` scales by a power of the floating-point radix.

$$\mathrm{ldexp}\, x\, e := x \cdot 2^e,$$
$$\mathrm{scalbn}\, x\, e := x \cdot r^e.$$

In MIT/GNU Scheme, these procedures are the same; they are both provided to make it clearer which operation is meant.

`flo:logb` *x* [procedure]

For nonzero finite x, returns $\lfloor \log_r x \rfloor$ as an exact integer, where r is the floating-point radix.

For all other inputs, raises invalid-operation and returns `#f`.

`flo:nextafter` *x1 x2* [procedure]

Returns the next floating-point number after *x1* in the direction of *x2*.

```
(flo:nextafter 0. -1.)          ⇒  -4.9406564584124654e-324
```

`flo:radix` [constant]
`flo:radix.` [constant]
`flo:precision` [constant]

Floating-point system parameters. `Flo:radix` is the floating-point radix as an integer, and `flo:precision` is the floating-point precision as an integer; `flo:radix.` is the flotaing-point radix as a flonum.

`flo:error-bound` [constant]
`flo:log-error-bound` [constant]
`flo:ulp-of-one` [constant]

`flo:log-ulp-of-one` [constant]

`Flo:error-bound`, sometimes called the machine epsilon, is the maximum relative error of rounding to nearest:

$$\max_x \frac{|x - \mathrm{fl}(x)|}{|x|} = \frac{1}{2r^{p-1}},$$

where r is the floating-point radix and p is the floating-point precision.

`Flo:ulp-of-one` is the distance from 1 to the next larger floating-point number, and is equal to $1/r^{p-1}$.

`Flo:error-bound` is half `flo:ulp-of-one`.

`Flo:log-error-bound` is the logarithm of `flo:error-bound`, and `flo:log-ulp-of-one` is the logarithm of `flo:log-ulp-of-one`.

`flo:ulp` *flonum* [procedure]

Returns the distance from *flonum* to the next floating-point number larger in magnitude with the same sign. For zero, this returns the smallest subnormal. For infinities, this returns positive infinity. For NaN, this returns the same NaN.

```
(flo:ulp 1.)                      ⇒  2.220446049250313e-16
(= (flo:ulp 1.) flo:ulp-of-one) ⇒  #t
```

`flo:normal-exponent-max` [constant]
`flo:normal-exponent-min` [constant]
`flo:subnormal-exponent-min` [constant]

Largest and smallest positive integer exponents of the radix in normal and subnormal floating-point numbers.

- `Flo:normal-exponent-max` is the largest positive integer such that `(expt flo:radix. flo:normal-exponent-max)` does not overflow.
- `Flo:normal-exponent-min` is the smallest positive integer such that `(expt flo:radix. flo:normal-exponent-min)` is a normal floating-point number.
- `Flo:subnormal-exponent-min` is the smallest positive integer such that `(expt flo:radix. flo:subnormal-exponent-min)` is nonzero; this is also the smallest positive floating-point number.

`flo:largest-positive-normal` [constant]
`flo:smallest-positive-normal` [constant]
`flo:smallest-positive-subnormal` [constant]

Smallest and largest normal and subnormal numbers in magnitude.

`flo:greatest-normal-exponent-base-e` [constant]
`flo:greatest-normal-exponent-base-2` [constant]
`flo:greatest-normal-exponent-base-10` [constant]
`flo:least-normal-exponent-base-e` [constant]
`flo:least-normal-exponent-base-2` [constant]
`flo:least-normal-exponent-base-10` [constant]
`flo:least-subnormal-exponent-base-e` [constant]
`flo:least-subnormal-exponent-base-2` [constant]

`flo:least-subnormal-exponent-base-10` [constant]

Least and greatest exponents of normal and subnormal floating-point numbers, as floating-point numbers. For example, `flo:greatest-normal-exponent-base-2` is the greatest floating-point number such that `(expt 2. flo:greatest-normal-exponent-base-2)` does not overflow and is a normal floating-point number.

`flo:total<` *x1 x2* [procedure]
`flo:total-mag<` *x1 x2* [procedure]
`flo:total-order` *x1 x2* [procedure]
`flo:total-order-mag` *x1 x2* [procedure]

These procedures implement the IEEE 754-2008 total ordering on floating-point values and their magnitudes. Here the "magnitude" of a floating-point value is a floating-point value with positive sign bit and everything else the same; e.g., `+nan.123` is the "magnitude" of `-nan.123` and `0.0` is the "magnitude" of `-0.0`.

The total ordering has little to no numerical meaning and should be used only when an arbitrary choice of total ordering is required for some non-numerical reason.

- `Flo:total<` returns true if *x1* precedes *x2*.
- `Flo:total-mag<` returns true if the magnitude of *x1* precedes the magnitude of *x2*.
- `Flo:total-order` returns -1 if *x1* precedes *x2*, 0 if they are the same floating-point value (including sign of zero, or sign and payload of NaN), and $+1$ if *x1* follows *x2*.
- `Flo:total-order-mag` returns -1 if the magnitude of *x1* precedes the magnitude of *x2*, etc.

`flo:make-nan` *negative? quiet? payload* [procedure]
`flo:nan-quiet?` *nan* [procedure]
`flo:nan-payload` *nan* [procedure]

`Flo:make-nan` creates a NaN given the sign bit, quiet bit, and payload. *Negative?* and *quiet?* must be booleans, and *payload* must be an unsigned $(p-2)$-bit integer, where p is the floating-point precision. If *quiet?* is false, *payload* must be nonzero.

```
(flo:sign-negative? (flo:make-nan negative? quiet? payload))
                                  ⇒  negative?
(flo:nan-quiet? (flo:make-nan negative? quiet? payload))
                                  ⇒  quiet?
(flo:nan-payload (flo:make-nan negative? quiet? payload))
                                  ⇒  payload

(flo:make-nan #t #f 42)           ⇒  -snan.42
(flo:sign-negative? +nan.123)     ⇒  #f
(flo:nan-quiet? +nan.123)         ⇒  #t
(flo:nan-payload +nan.123)        ⇒  123
```

4.8.3 Floating-Point Environment

The IEEE 754-2008 computation model includes a persistent rounding mode, exception flags, and exception-handling modes. In MIT/GNU Scheme, the floating-point environment is

per-thread. However, because saving and restoring the floating-point environment is expensive, it is maintained only for those threads that have touched the floating-point environment explicitly, either:

- during a procedure such as `flo:with-exceptions-trapped` that establishes a change to the floating-point environment for a dynamic extent, or
- after `flo:set-environment!` to a non-default environment (but not after `flo:set-environment!` to the default environment), or
- after various other procedures such as `flo:clear-exceptions!` that explicitly change the floating-point environment.

The default environment is as in IEEE 754-2008: no exceptions are trapped, and rounding is to nearest with ties broken to even. The set of exception flags in the default environment is indeterminate — callers must enter a per-thread environment, e.g. by calling `flo:clear-exceptions!`, before acting on the exception flags. Like the default environment, a per-thread environment initially has no exceptions trapped and rounds to nearest with ties to even.

A **floating-point environment** descriptor is a machine-dependent object representing the IEEE 754-2008 floating-point rounding mode, exception flags, and exception-handling mode. Users should not inspect a floating-point environment descriptor other than to use it with the procedures here; its representation may vary from system to system.

`flo:default-environment` [procedure]
: Returns a descriptor for the default environment, with no exceptions trapped and round-to-nearest/ties-to-even.

`flo:with-default-environment` *thunk* [procedure]
: Calls *thunk* in the default floating-point environment, and restores the caller's floating-point environment afterward. Equivalent to:

```
(flo:preserving-environment
 (lambda ()
   (flo:set-environment! (flo:default-environment))
   (thunk)))
```

`flo:environment` [procedure]
`flo:set-environment!` *floenv* [procedure]
`flo:update-environment!` *floenv* [procedure]
: `Flo:environment` returns a descriptor for the current floating-point environment. `Flo:set-environment!` replaces the current floating-point environment by *floenv*. `Flo:update-environment!` does likewise, but re-raises any exceptions that were already raised in the current floating-point environment, which may cause a trap if *floenv* also traps them.

  `Flo:update-environment!` is usually used together with `flo:defer-exception-traps!` to defer potentially trapping on exceptions in a large intermediate computation until the end.

`flo:preserving-environment` *thunk* [procedure]
: Saves the current floating-point environment if any and calls *thunk*. On exit from *thunk*, including non-local exit, saves *thunk*'s floating-point environment and restores

the original floating-point environment as if with `flo:set-environment!`. On re-entry into *thunk*, restores *thunk*'s floating-point environment.

Note: `Flo:preserving-environment` *does not* enter a per-thread environment. If the current thread is in the default environment, the exception flags are indeterminate, and remain so inside `flo:preserving-environment`. Callers interested in using the exception flags should start inside `flo:preserving-environment` by clearing them with `flo:clear-exceptions!`.

4.8.4 Floating-Point Exceptions

In IEEE 754-2008, floating-point operations such as arithmetic may raise exceptions. This sets a flag in the floating-point environment that is maintained until it is cleared. Many machines can also be configured to trap on exceptions, which in Scheme leads to signalling a condition. (Not all CPUs support trapping exceptions — for example, most ARMv8 CPUs do not.) In the default environment, no exceptions are trapped.

Floating-point exceptions and sets of floating-point exceptions are represented by small integers, whose interpretation is machine-dependent — for example, the invalid-operation exception may be represented differently on PowerPC and AMD x86-64 CPUs. The number for a floating-point exception is the same as the number for a set of exceptions containing only that one; the bitwise-AND of two sets is their intersection, the bitwise-IOR is their union, etc. The procedures `flo:exceptions->names` and `flo:names->exceptions` convert between machine-dependent integer representations and machine-independent lists of human-readable symbols.

The following exceptions are recognized by MIT/GNU Scheme:

inexact-result
: Raised when the result of a floating-point computation is not a floating-point number and therefore must be rounded.

  The inexact-result exception is never trappable in MIT/GNU Scheme.

underflow
: Raised when the result of a floating-point computation is too small in magnitude to be represented by a normal floating-point number, and is therefore rounded to a subnormal or zero.

overflow
: Raised when the result of a floating-point computation is too large in magnitude to be represented by a floating-point number, and is therefore rounded to infinity.

divide-by-zero
: Raised on division of a nonzero finite real number by a zero real number, or logarithm of zero, or other operation that has an unbounded limit at a point like division by a divisor approaching zero.

invalid-operation
: Raised when the input to a floating-point computation is nonsensical, such as division of zero by zero, or real logarithm of a negative number. The result of an invalid-operation is a NaN. Also raised when the input to a floating-point operation is a signalling NaN, but not for a quiet NaN.

subnormal-operand
: Raised when an operand in a floating-point operation is subnormal.

(This is not a standard IEEE 754-2008 exception. It is supported by Intel CPUs.)

`flo:supported-exceptions` [procedure]
Returns the set of exceptions that are supported on the current machine.

`flo:trappable-exceptions` [procedure]
Returns the set of exceptions that can be trapped on the current machine. This is usually the same as the supported exceptions, except for inexact-result, which is never trappable in MIT/GNU Scheme.

`flo:exception:divide-by-zero` [procedure]
`flo:exception:inexact-result` [procedure]
`flo:exception:invalid-operation` [procedure]
`flo:exception:overflow` [procedure]
`flo:exception:subnormal-operand` [procedure]
`flo:exception:underflow` [procedure]
Returns the specified floating-point exception number. On machines that do not support a particular exception, the corresponding procedure simply returns `0`.

`flo:exceptions->names` *excepts* [procedure]
`flo:names->exceptions` *list* [procedure]
These procedures convert between a machine-dependent small integer representation of a set of exceptions, and a representation of a set of exceptions by a list of human-readable symbols naming them.

```
(flo:preserving-environment
 (lambda ()
   (flo:clear-exceptions! (flo:supported-exceptions))
   (flo:/ (identity-procedure 1.) 0.)
   (flo:exceptions->names
    (flo:test-exceptions (flo:supported-exceptions)))))
                                ⇒  (divide-by-zero)
```

`flo:test-exceptions` *excepts* [procedure]
Returns the set of exceptions in *excepts* that are currently raised.

In the default environment, the result is indeterminate, and may be affected by floating-point operations in other threads.

`flo:clear-exceptions!` *excepts* [procedure]
`flo:raise-exceptions!` *excepts* [procedure]
Clears or raises the exceptions in *excepts*, entering a per-thread environment. Other exceptions are unaffected.

`flo:save-exception-flags` [procedure]
`flo:restore-exception-flags!` *exceptflags* [procedure]
`flo:test-exception-flags` *exceptflags excepts* [procedure]
`Flo:save-exception-flags` returns a machine-dependent representation of the currently trapped and raised exceptions. `Flo:restore-exception-flags!` restores it, entering a per-thread environment. `Flo:test-exception-flags` returns the set of exceptions in *excepts* that are raised in *exceptflags*.

Exceptflags is *not* the same as a set of exceptions. It is opaque and machine-dependent and should not be used except with `flo:restore-exception-flags!` and `flo:test-exception-flags`.

Bug: `Flo:test-exception-flags` is unimplemented.

`flo:have-trap-enable/disable?` [procedure]
Returns true if trapping floating-point exceptions is supported on this machine.

`flo:default-trapped-exceptions` [procedure]
Returns the set of exceptions that are trapped in the default floating-point environment. Equivalent to `(flo:names->exceptions '())`, or simply `0`, since by default, no exceptions are trapped.

`flo:trapped-exceptions` [procedure]
Returns the set of exceptions that are currently trapped.

`flo:trap-exceptions!` *excepts* [procedure]
`flo:untrap-exceptions!` *excepts* [procedure]
`flo:set-trapped-exceptions!` *excepts* [procedure]
`Flo:trap-exceptions!` requests that any exceptions in the set *excepts* be trapped, in addition to all of the ones that are currently trapped. `Flo:untrap-exceptions!` requests that any exceptions in the set *excepts* not be trapped. `Flo:set-trapped-exceptions!` replaces the set of trapped exceptions altogether by *excepts*. All three procedures enter a per-thread environment.

```
(define (flo:trap-exceptions! excepts)
  (flo:set-trapped-exceptions!
   (fix:or (flo:trapped-exceptions) excepts)))

(define (flo:untrap-exceptions! excepts)
  (flo:set-trapped-exceptions!
   (fix:andc (flo:trapped-exceptions) excepts)))

(define (flo:set-trapped-exceptions! excepts)
  (flo:trap-exceptions! excepts)
  (flo:untrap-exceptions!
   (fix:andc (flo:supported-exceptions) excepts)))
```

`flo:with-exceptions-trapped` *excepts thunk* [procedure]
`flo:with-exceptions-untrapped` *excepts thunk* [procedure]
`flo:with-trapped-exceptions` *excepts thunk* [procedure]
Dynamic-extent analogues of `flo:trap-exceptions!`, `flo:untrap-exceptions!`, and `flo:set-trapped-exceptions!`. These call *thunk* with their respective changes to the set of trapped exceptions in a per-thread environment, and restore the environment on return or non-local exit.

`flo:defer-exception-traps!` [procedure]
Saves the current floating-point environment, clears all raised exceptions, disables all exception traps, and returns a descriptor for the saved floating-point environment.

`Flo:defer-exception-traps!` is typically used together with `flo:update-environment!`, to trap any exceptions that the caller had wanted trapped only after a long intermediate computation. This pattern is captured in `flo:deferring-exception-traps`.

`flo:deferring-exception-traps` *thunk* [procedure]
Calls *thunk*, but defers trapping on any exceptions it raises until it returns. Equivalent to:

```
(flo:preserving-environment
 (lambda ()
   (let ((environment (flo:defer-exception-traps!)))
     (begin0 (thunk)
       (flo:update-environment! environment)))))
```

`flo:ignoring-exception-traps` *thunk* [procedure]
Calls *thunk* with all exceptions untrapped and unraised. Equivalent to:

```
(flo:preserving-environment
 (lambda ()
   (flo:defer-exception-traps!)
   (thunk)))
```

4.8.5 Floating-Point Rounding Mode

IEEE 754-2008 supports four rounding modes, which determine the answer given by a floating-point computation when the exact result lies between two floating-point numbers but is not a floating-point number itself:

`to-nearest`
: Round to the nearest floating-point number. If there are two equidistant ones, choose the one whose least significant digit is even. Also known as "round-to-nearest/ties-to-even".

`toward-zero`
: Round to the floating-point number closest to zero.

`downward` Round to the greatest floating-point number below.

`upward` Round to the least floating-point number above.

Warning: Not all procedures in MIT/GNU Scheme respect the rounding mode. Only the basic arithmetic operations — +, −, ∗, /, and sqrt — will reliably respect it. The main purpose of changing the rounding mode is to diagnose numerical instability by injecting small perturbations throughout the computation.

Bug: It would be nice if we had "round-to-odd", where any inexact result is rounded to the nearest *odd* floating-point number, for implementing "doubled"-precision algorithms. But we don't. Sorry.

`flo:default-rounding-mode` [procedure]
Returns a symbol for the default rounding mode, which is always `to-nearest`.

`flo:rounding-modes` [procedure]
Returns a list of the supported rounding modes as symbols.

`flo:rounding-mode` [procedure]
`flo:set-rounding-mode!` *mode* [procedure]
Gets or sets the current rounding mode as a symbol, entering a per-thread environment.

`flo:with-rounding-mode` *mode thunk* [procedure]
Call *thunk* in a per-thread environment with the rounding mode set to *mode*. On return, the floating-point environment, including rounding mode, is restored to what it was before.

Non-local exit from and re-entrance to *thunk* behaves as if the call is surrounded by `flo:preserving-environment` (see Section 4.8.3 [Floating-Point Environment], page 98).

4.9 Random Number Generation

MIT/GNU Scheme provides a facility for random number generation. The current implementation uses the ChaCha stream cipher, reseeding itself at each request so that past outputs cannot be distinguished from uniform random even if the state of memory is compromised in the future.

The interface described here is a mixture of the Common Lisp and SRFI 27 systems.

`random` *m* [*state*] [procedure]
The argument *m* must be either an exact positive integer, or an inexact positive real.

- If *m* is an exact positive integer, then `random` returns an exact nonnegative integer below *m* with uniform distribution.
- If *m* is an inexact positive real, then `random` returns an inexact real in the interval $[0, m)$ with uniform distribution.

If *state* is given and not `#f`, it must be a random-state object; otherwise, it defaults to the `default-random-source`. This object is used to maintain the state of the pseudorandom number generator and is altered as a side effect of the `random` procedure.

Use of the default random state requires synchronization between threads, so it is better for multithreaded programs to use explicit states.

```
(random 1.0)    ⇒  .32744744667719056
(random 1.0)    ⇒  .01668326768172354
(random 10)     ⇒  3
(random 10)     ⇒  8
(random 100)    ⇒  38
(random 100)    ⇒  63
```

`flo:random-unit-closed` *state* [procedure]
`flo:random-unit-open` *state* [procedure]
State must be a random-state object. `Flo:random-unit-closed` returns a flonum in the closed interval $[0, 1]$ with uniform distribution. In practical terms, the result is in the half-closed interval $(0, 1]$ because the probability of returning 0 is 2^{-1075}, far below the standard probability 2^{-128} that means "never" in cryptographic engineering terms.

`Flo:random-unit-open` is like `flo:random-unit-closed`, but it explicitly rejects `0.0` and `1.0` as outputs, so that the result is a floating-point number in the open interval $(0, 1)$. `(flo:random-unit-open)` is equivalent `(random 1.)`, except that it is faster.

Callers should generally use `flo:random-unit-closed`, because for the uniform distribution on the interval $[0, 1]$ of real numbers, the probability of a real number that is rounded to the floating-point `1.0` is the small but nonnegligible 2^{-54}, and arithmetic downstream should be prepared to handle results that are rounded to `1.0` much more readily than results that are rounded to `0.0` — in other words, a requirement to use `flo:random-unit-open` is evidence of bad numerics downstream.

`flo:random-unit` *state* [procedure]
Deprecated alias for `flo:random-unit-open`.

`random-bytevector` *n* [*state*] [procedure]
Returns a bytevector of *n* bytes drawn independently uniformly at random from *state*.

`random-bytevector!` *bytevector* [*start end state*] [procedure]
Replaces the bytes in *bytevector* from *start* to *end* by bytes drawn independently uniformly at random from *state*.

The next three definitions concern random-state objects. In addition to these definitions, it is important to know that random-state objects are specifically designed so that they can be saved to disk using the `fasdump` procedure, and later restored using the `fasload` procedure. This allows a particular random-state object to be saved in order to replay a particular pseudorandom sequence.

`*random-state*` [variable]
This variable is **deprecated**; pass an explicit state instead.

`make-random-state` [*state*] [procedure]
This procedure returns a new random-state object, suitable for use as as the *state* argument to `random`. If *state* is not given or `#f`, `make-random-state` returns a *copy* of `default-random-source`. If *state* is a random-state object, a copy of that object is returned. If *state* is `#t`, then a new random-state object is returned that has been "randomly" initialized by some means (such as by a time-of-day clock).

`random-state?` *object* [procedure]
Returns `#t` if *object* is a random-state object, otherwise returns `#f`.

`export-random-state` *state* [procedure]
`import-random-state` *state* [procedure]
`Export-random-state` returns an external representation of a random state — an object that can be safely read and written with `read` and `write`, consisting only of nested lists, vectors, symbols, and small exact integers. `Import-random-state` creates a random state from its external representation.

In the MIT/GNU Scheme implementation of the SRFI 27 API, a "random source" happens to be the same as a random state, but users should not rely on this.

`make-random-source` [procedure]
[SRFI 27] Returns a random source. Every random source created by `make-random-source` returns the same sequence of outputs unless modified by `random-source-state-set!`, `random-source-randomize!`, or `random-source-pseudo-randomize!`.

`random-source?` *object* [procedure]
[SRFI 27] Returns `#t` if *object* is a random source, otherwise returns `#f`.

`default-random-source` [constant]
[SRFI 27] The default random source, used by the various random procedures if no explicit state is specified and `*random-state*` is false.

`random-source-state-ref` *source* [procedure]
`random-source-state-set!` *source exported-state* [procedure]
[SRFI 27] `Random-source-state-ref` returns an external representation of a random source — an object that can be safely read and written with `read` and `write`, consisting only of nested lists, vectors, symbols, and small exact integers. `Random-source-state-set!` replaces the innards of *source* by the source represented by *exported-state* from `random-source-state-ref`.

`random-source-randomize!` *source* [procedure]
[SRFI 27] Loads entropy from the environment into *source* so that its subsequent outputs are nondeterministic.

Warning: Most implementations of SRFI 27 *do not* make subsequent outputs unpredictable with cryptography, so don't rely on this.

`random-source-pseudo-randomize!` *source i j* [procedure]
[SRFI 27] The arguments i and j must be exact nonnegative integers below 2^{128}. This procedure sets *source* to generate one of 2^{256} distinct possible streams of output, so that if i and j are chosen uniformly at random, it is hard to distinguish the outputs of the source from uniform random.

Warning: Most implementations of SRFI 27 *do not* make it hard to distinguish the outputs of the source from uniform random even if the indices i and j are uniform random, so don't rely on this.

`random-integer` *n* [procedure]
[SRFI 27] Returns an exact nonnegative integer below n chosen uniformly at random.

Equivalent to:

```
((random-source-make-integers default-random-source) n)
```

`random-real` [procedure]
[SRFI 27] Returns an inexact real in the open interval $(0, 1)$ with uniform distribution.

Equivalent to:

```
((random-source-make-reals default-random-source))
```

`random-source-make-integers` *source* [procedure]
[SRFI 27] Returns a procedure of one argument, n, that deterministically draws from *source* an exact nonnegative integer below n with uniform distribution.

`random-source-make-reals` *source* [*unit*] [procedure]

[SRFI 27] Returns a procedure of zero arguments that deterministically draws from *source* an inexact real in the interval $(0, 1)$ with uniform distribution. If *unit* is specified, the results are instead uniform random integral multiples of *unit* in $(0, 1)$ and of the same exactness as *unit*.

5 Characters

Characters are objects that represent printed characters such as letters and digits. MIT/GNU Scheme supports the full Unicode character repertoire.

Characters are written using the notation `#\`*`character`* or `#\`*`character-name`* or `#\x`*`hex-scalar-value`*.

The following standard character names are supported:

```
#\alarm                 ; U+0007
#\backspace             ; U+0008
#\delete                ; U+007F
#\escape                ; U+001B
#\newline               ; the linefeed character, U+000A
#\null                  ; the null character, U+0000
#\return                ; the return character, U+000D
#\space                 ; the preferred way to write a space, U+0020
#\tab                   ; the tab character, U+0009
```

Here are some additional examples:

```
#\a                     ; lowercase letter
#\A                     ; uppercase letter
#\(                     ; left parenthesis
#\                      ; the space character
```

Case is significant in `#\`*`character`*, and in `#\`*`character-name`*, but not in `#\x`*`hex-scalar-value`*. If *character* in `#\`*`character`* is alphabetic, then any character immediately following *character* cannot be one that can appear in an identifier. This rule resolves the ambiguous case where, for example, the sequence of characters '`#\space`' could be taken to be either a representation of the space character or a representation of the character '`#\s`' followed by a representation of the symbol '`pace`'.

Characters written in the `#\` notation are self-evaluating. That is, they do not have to be quoted in programs.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have '`-ci`' (for "case insensitive") embedded in their names.

MIT/GNU Scheme allows a character name to include one or more *bucky bit* prefixes to indicate that the character includes one or more of the keyboard shift keys Control, Meta, Super, or Hyper (note that the Control bucky bit prefix is not the same as the ASCII control key). The bucky bit prefixes and their meanings are as follows (case is not significant):

```
Key             Bucky bit prefix        Bucky bit
---             ----------------        ---------

Meta            M- or Meta-                1
Control         C- or Control-             2
Super           S- or Super-               4
Hyper           H- or Hyper-               8
```

For example,

```
#\c-a                   ; Control-a
#\meta-b                ; Meta-b
#\c-s-m-h-A             ; Control-Meta-Super-Hyper-A
```

`char->name` *char* [procedure]

Returns a string corresponding to the printed representation of *char*. This is the *character*, *character-name*, or `xhex-scalar-value` component of the external representation, combined with the appropriate bucky bit prefixes.

```
(char->name #\a)                ⇒  "a"
(char->name #\space)            ⇒  "space"
(char->name #\c-a)              ⇒  "C-a"
(char->name #\control-a)        ⇒  "C-a"
```

`name->char` *string* [procedure]

Converts a string that names a character into the character specified. If *string* does not name any character, `name->char` signals an error.

```
(name->char "a")                ⇒  #\a
(name->char "space")            ⇒  #\space
(name->char "SPACE")            ⇒  #\space
(name->char "c-a")              ⇒  #\C-a
(name->char "control-a")        ⇒  #\C-a
```

`char?` *object* [standard procedure]

Returns `#t` if *object* is a character, otherwise returns `#f`.

`char=?` *char1 char2 char3 . . .* [standard procedure]
`char<?` *char1 char2 char3 . . .* [standard procedure]
`char>?` *char1 char2 char3 . . .* [standard procedure]
`char<=?` *char1 char2 char3 . . .* [standard procedure]
`char>=?` *char1 char2 char3 . . .* [standard procedure]

These procedures return `#t` if the results of passing their arguments to `char->integer` are respectively equal, monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing.

These predicates are transitive.

`char-ci=?` *char1 char2 char3 . . .* [char library procedure]
`char-ci<?` *char1 char2 char3 . . .* [char library procedure]
`char-ci>?` *char1 char2 char3 . . .* [char library procedure]
`char-ci<=?` *char1 char2 char3 . . .* [char library procedure]
`char-ci>=?` *char1 char2 char3 . . .* [char library procedure]

These procedures are similar to `char=?` et cetera, but they treat upper case and lower case letters as the same. For example, `(char-ci=? #\A #\a)` returns `#t`.

Specifically, these procedures behave as if `char-foldcase` were applied to their arguments before they were compared.

`char-alphabetic?` *char* [char library procedure]
`char-numeric?` *char* [char library procedure]

`char-whitespace?` *char* [char library procedure]
`char-upper-case?` *char* [char library procedure]
`char-lower-case?` *char* [char library procedure]

These procedures return `#t` if their arguments are alphabetic, numeric, whitespace, upper case, or lower case characters respectively, otherwise they return `#f`.

Specifically, they return `#t` when applied to characters with the Unicode properties Alphabetic, Numeric_Decimal, White_Space, Uppercase, or Lowercase respectively, and `#f` when applied to any other Unicode characters. Note that many Unicode characters are alphabetic but neither upper nor lower case.

`char-alphanumeric?` *char* [procedure]

Returns `#t` if *char* is either alphabetic or numeric, otherwise it returns `#f`.

`digit-value` *char* [char library procedure]

This procedure returns the numeric value (0 to 9) of its argument if it is a numeric digit (that is, if `char-numeric?` returns `#t`), or `#f` on any other character.

```
(digit-value #\3) ⇒ 3
(digit-value #\x0664) ⇒ 4
(digit-value #\x0AE6) ⇒ 0
(digit-value #\x0EA6) ⇒ #f
```

`char->integer` *char* [standard procedure]
`integer->char` *n* [standard procedure]

Given a Unicode character, `char->integer` returns an exact integer between `0` and `#xD7FF` or between `#xE000` and `#x10FFFF` which is equal to the Unicode scalar value of that character. Given a non-Unicode character, it returns an exact integer greater than `#x10FFFF`.

Given an exact integer that is the value returned by a character when `char->integer` is applied to it, `integer->char` returns that character.

Implementation note: MIT/GNU Scheme allows any Unicode code point, not just scalar values.

Implementation note: If the argument to `char->integer` or `integer->char` is a constant, the MIT/GNU Scheme compiler will constant-fold the call, replacing it with the corresponding result. This is a very useful way to denote unusual character constants or ASCII codes.

`char-upcase` *char* [char library procedure]
`char-downcase` *char* [char library procedure]
`char-foldcase` *char* [char library procedure]

The `char-upcase` procedure, given an argument that is the lowercase part of a Unicode casing pair, returns the uppercase member of the pair. Note that language-sensitive casing pairs are not used. If the argument is not the lowercase member of such a pair, it is returned.

The `char-downcase` procedure, given an argument that is the uppercase part of a Unicode casing pair, returns the lowercase member of the pair. Note that language-sensitive casing pairs are not used. If the argument is not the uppercase member of such a pair, it is returned.

The `char-foldcase` procedure applies the Unicode simple case-folding algorithm to its argument and returns the result. Note that language-sensitive folding is not used. See UAX #44 (part of the Unicode Standard) for details.

Note that many Unicode lowercase characters do not have uppercase equivalents.

`char->digit` *char* [*radix*] [procedure]

If *char* is a character representing a digit in the given *radix*, returns the corresponding integer value. If *radix* is specified (which must be an exact integer between 2 and 36 inclusive), the conversion is done in that base, otherwise it is done in base 10. If *char* doesn't represent a digit in base *radix*, `char->digit` returns `#f`.

Note that this procedure is insensitive to the alphabetic case of *char*.

```
(char->digit #\8)                ⇒  8
(char->digit #\e 16)             ⇒  14
(char->digit #\e)                ⇒  #f
```

`digit->char` *digit* [*radix*] [procedure]

Returns a character that represents *digit* in the radix given by *radix*. The *radix* argument, if given, must be an exact integer between 2 and 36 (inclusive); it defaults to 10. The *digit* argument must be an exact non-negative integer strictly less than *radix*.

```
(digit->char 8)                  ⇒  #\8
(digit->char 14 16)              ⇒  #\E
```

5.1 Character implementation

An MIT/GNU Scheme character consists of a *code* part and a *bucky bits* part. The code part is a Unicode code point, while the bucky bits are an additional set of bits representing shift keys available on some keyboards.

There are 4 bucky bits, named *control*, *meta*, *super*, and *hyper*. On GNU/Linux systems running a graphical desktop, the control bit corresponds to the `CTRL` key; the meta bit corresponds to the `ALT` key; and the super bit corresponds to the "windows" key. On macOS, these are the `CONTROL`, `OPTION`, and `COMMAND` keys respectively.

Characters with bucky bits are not used much outside of graphical user interfaces (e.g. Edwin). They cannot be stored in strings or character sets, and aren't read or written by textual I/O ports.

`make-char` *code bucky-bits* [procedure]

Builds a character from *code* and *bucky-bits*. The value of *code* must be a Unicode code point; the value of *bucky-bits* must be an exact non-negative integer strictly less than `16`. If `0` is specified for *bucky-bits*, `make-char` produces an ordinary character; otherwise, the appropriate bits are set as follows:

```
1               meta
2               control
4               super
8               hyper
```

For example,

```
(make-char 97 0)                ⇒  #\a
(make-char 97 1)                ⇒  #\M-a
(make-char 97 2)                ⇒  #\C-a
(make-char 97 3)                ⇒  #\C-M-a
```

`char-code` *char* [procedure]

Returns the Unicode code point of *char*. Note that if *char* has no bucky bits set, then this is the same value returned by `char->integer`.

For example,

```
(char-code #\a)                 ⇒  97
(char-code #\c-a)               ⇒  97
```

`char-bits` *char* [procedure]

Returns the exact integer representation of *char*'s bucky bits. For example,

```
(char-bits #\a)                 ⇒  0
(char-bits #\m-a)               ⇒  1
(char-bits #\c-a)               ⇒  2
(char-bits #\c-m-a)             ⇒  3
```

`char-code-limit` [constant]

This constant is the strict upper limit on a character's *code* value. It is `#x110000` unless some future version of Unicode increases the range of code points.

`char-bits-limit` [constant]

This constant is the strict upper limit on a character's *bucky-bits* value. It is currently `#x10` and unlikely to change in the future.

`bitless-char?` *object* [procedure]

Returns `#t` if *object* is a character with no bucky bits set, otherwise it returns `#f` .

`char->bitless-char` *char* [procedure]

Returns *char* with any bucky bits removed. The result is guaranteed to satisfy `bitless-char?`.

`char-predicate` *char* [procedure]

Returns a procedure of one argument that returns `#t` if its argument is a character `char=?` to *char*, otherwise it returns `#f`.

`char-ci-predicate` *char* [procedure]

Returns a procedure of one argument that returns `#t` if its argument is a character `char-ci=?` to *char*, otherwise it returns `#f`.

5.2 Unicode

MIT/GNU Scheme implements the full Unicode character repertoire, defining predicates for Unicode characters and their associated integer values. A *Unicode code point* is an exact non-negative integer strictly less than `#x110000`. A *Unicode scalar value* is a Unicode code point that doesn't fall between `#xD800` inclusive and `#xE000` exclusive; in other words, any Unicode code point except for the *surrogate* code points.

`unicode-code-point?` *object* [procedure]
Returns `#t` if *object* is a Unicode code point, otherwise it returns `#f`.

`unicode-scalar-value?` *object* [procedure]
Returns `#t` if *object* is a Unicode scalar value, otherwise it returns `#f`.

`unicode-char?` *object* [procedure]
Returns `#t` if *object* is any character corresponding to a Unicode code point, except for those with general category `other:surrogate` or `other:not-assigned`.

`char-general-category` *char* [procedure]
`code-point-general-category` *code-point* [procedure]
Returns the Unicode general category of *char* (or *code-point*) as a descriptive symbol:

Category	**Symbol**
Lu	`letter:uppercase`
Ll	`letter:lowercase`
Lt	`letter:titlecase`
Lm	`letter:modifier`
Lo	`letter:other`
Mn	`mark:nonspacing`
Mc	`mark:spacing-combining`
Me	`mark:enclosing`
Nd	`number:decimal-digit`
Nl	`number:letter`
No	`number:other`
Pc	`punctuation:connector`
Pd	`punctuation:dash`
Ps	`punctuation:open`
Pe	`punctuation:close`
Pi	`punctuation:initial-quote`
Pf	`punctuation:final-quote`
Po	`punctuation:other`
Sm	`symbol:math`
Sc	`symbol:currency`
Sk	`symbol:modifier`
So	`symbol:other`
Zs	`separator:space`
Zl	`separator:line`
Zp	`separator:paragraph`
Cc	`other:control`
Cf	`other:format`
Cs	`other:surrogate`
Co	`other:private-use`
Cn	`other:not-assigned`

5.3 Character Sets

MIT/GNU Scheme's character-set abstraction is used to represent groups of characters, such as the letters or digits. A character set may contain any character. Alternatively, a character set can be treated as a set of code points.

Implementation note: MIT/GNU Scheme allows any "bitless" character to be stored in a character set; operations that accept characters automatically strip their bucky bits.

`char-set?` *object* [procedure]
: Returns `#t` if *object* is a character set, otherwise it returns `#f`.

`char-in-set?` *char char-set* [procedure]
: Returns `#t` if *char* is in *char-set*, otherwise it returns `#f`.

`code-point-in-set?` *code-point char-set* [procedure]
: Returns `#t` if *code-point* is in *char-set*, otherwise it returns `#f`.

`char-set-predicate` *char-set* [procedure]
: Returns a procedure of one argument that returns `#t` if its argument is a character in *char-set*, otherwise it returns `#f`.

`compute-char-set` *predicate* [procedure]
: Calls *predicate* once on each Unicode code point, and returns a character set containing exactly the code points for which *predicate* returns a true value.

The next procedures represent a character set as a *code-point list*, which is a list of *code-point range* elements. A code-point range is either a Unicode code point, or a pair `(`*start* `.` *end*`)` that specifies a contiguous range of code points. Both *start* and *end* must be exact nonnegative integers less than or equal to `#x110000`, and *start* must be less than or equal to *end*. The range specifies all of the code points greater than or equal to *start* and strictly less than *end*.

`char-set` *element* ... [procedure]
`char-set*` *elements* [procedure]
: Returns a new character set consisting of the characters specified by *element*s. The procedure `char-set` takes these elements as multiple arguments, while `char-set*` takes them as a single list-valued argument; in all other respects these procedures are identical.

  An *element* can take several forms, each of which specifies one or more characters to include in the resulting character set: a character includes itself; a string includes all of the characters it contains; a character set includes its members; or a code-point range includes the corresponding characters.

  In addition, an *element* may be a symbol from the following table, which represents the characters as shown:

Name	**Unicode character specification**
`alphabetic`	Alphabetic = True
`alphanumeric`	Alphabetic = True | Numeric_Type = Decimal
`cased`	Cased = True
`lower-case`	Lowercase = True

`numeric`	Numeric_Type = Decimal
`unicode`	General_Category != (Cs | Cn)
`upper-case`	Uppercase = True
`whitespace`	White_Space = True

`char-set->code-points` *char-set* [procedure]
Returns a code-point list specifying the contents of *char-set*. The returned list consists of numerically sorted, disjoint, and non-abutting code-point ranges.

`char-set=?` *char-set-1 char-set-2* [procedure]
Returns `#t` if *char-set-1* and *char-set-2* contain exactly the same characters, otherwise it returns `#f`.

`char-set-invert` *char-set* [procedure]
Returns a character set that's the inverse of *char-set*. That is, the returned character set contains exactly those characters that aren't in *char-set*.

`char-set-union` *char-set ...* [procedure]
`char-set-intersection` *char-set ...* [procedure]
`char-set-difference` *char-set-1 char-set ...* [procedure]
These procedures compute the respective set union, set intersection, and set difference of their arguments.

`char-set-union*` *char-sets* [procedure]
`char-set-intersection*` *char-sets* [procedure]
These procedures correspond to `char-set-union` and `char-set-intersection` but take a single argument that's a list of character sets rather than multiple character-set arguments.

`char-set:alphabetic` [constant]
`char-set:numeric` [constant]
`char-set:whitespace` [constant]
`char-set:upper-case` [constant]
`char-set:lower-case` [constant]
`char-set:alphanumeric` [constant]
These constants are the character sets corresponding to `char-alphabetic?`, `char-numeric?`, `char-whitespace?`, `char-upper-case?`, `char-lower-case?`, and `char-alphanumeric?` respectively.

`8-bit-char-set?` *char-set* [procedure]
Returns `#t` if *char-set* contains only 8-bit code points (i.e.. ISO 8859-1 characters), otherwise it returns `#f`.

6 Strings

Strings are sequences of characters. Strings are written as sequences of characters enclosed within quotation marks ("). Within a string literal, various escape sequences represent characters other than themselves. Escape sequences always start with a backslash (\):

`\a` : alarm, U+0007
`\b` : backspace, U+0008
`\t` : character tabulation, U+0009
`\n` : linefeed, U+000A
`\r` : return, U+000D
`\"` : double quote, U+0022
`\\` : backslash, U+005C
`\|` : vertical line, U+007C
*intraline-whitespace** *line-ending* *intraline-whitespace**
: nothing
`\x`*hex-scalar-value*`;`
: specified character (note the terminating semi-colon).

The result is unspecified if any other character in a string occurs after a backslash.

Except for a line ending, any character outside of an escape sequence stands for itself in the string literal. A line ending which is preceded by *intraline-whitespace* expands to nothing (along with any trailing intraline whitespace), and can be used to indent strings for improved legibility. Any other line ending has the same effect as inserting a `\n` character into the string.

Examples:

```
"The word \"recursion\" has many meanings."
"Another example:\ntwo lines of text"
"Here's text \
   containing just one line"
"\x03B1; is named GREEK SMALL LETTER ALPHA."
```

The *length* of a string is the number of characters that it contains. This number is an exact, non-negative integer that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The names of the versions that ignore case end with '`-ci`' (for "case insensitive").

Implementations may forbid certain characters from appearing in strings. However, with the exception of `#\null`, ASCII characters must not be forbidden. For example, an implementation might support the entire Unicode repertoire, but only allow characters U+0001 to U+00FF (the Latin-1 repertoire without `#\null`) in strings.

Implementation note: MIT/GNU Scheme allows any "bitless" character to be stored in a string. In effect this means any character with a Unicode code point, including surrogates. String operations that accept characters automatically strip their bucky bits.

It is an error to pass such a forbidden character to `make-string`, `string`, `string-set!`, or `string-fill!`, as part of the list passed to `list->string`, or as part of the vector

passed to `vector->string`, or in UTF-8 encoded form within a bytevector passed to `utf8->string`. It is also an error for a procedure passed to `string-map` to return a forbidden character, or for `read-string` to attempt to read one.

MIT/GNU Scheme supports both *mutable* and *immutable* strings. Procedures that mutate strings, in particular `string-set!` and `string-fill!`, will signal an error if given an immutable string. Nearly all procedures that return strings return immutable strings; notable exceptions are `make-string` and `string-copy`, which always return mutable strings, and `string-builder` which gives the programmer the ability to choose mutable or immutable results.

`string?` *obj* [standard procedure]
Returns `#t` if *obj* is a string, otherwise returns `#f`.

`make-string` *k* [*char*] [standard procedure]
The `make-string` procedure returns a newly allocated mutable string of length *k*. If *char* is given, then all the characters of the string are initialized to *char*, otherwise the contents of the string are unspecified.

`string` *object ...* [extended standard procedure]
`string*` *objects* [procedure]
Returns an immutable string whose characters are the concatenation of the characters from the given objects. Each object is converted to characters as if passed to the `display` procedure.

This is an MIT/GNU Scheme extension to the standard `string` that accepts only characters as arguments.

The procedure `string*` is identical to `string` but takes a single argument that's a list of objects, rather than multiple object arguments.

`string-length` *string* [standard procedure]
Returns the number of characters in the given *string*.

`string-ref` *string k* [standard procedure]
It is an error if *k* is not a valid index of *string*.

The `string-ref` procedure returns character *k* of *string* using zero-origin indexing. There is no requirement for this procedure to execute in constant time.

`string-set!` *string k char* [standard procedure]
It is an error if `string` is not a mutable string or if *k* is not a valid index of *string*.

The `string-set!` procedure stores *char* in element *k* of *string*. There is no requirement for this procedure to execute in constant time.

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)  ⇒  unspecified
(string-set! (g) 0 #\?)  ⇒  error
(string-set! (symbol->string 'immutable) 0 #\?)  ⇒  error
```

`string=?` *string1 string2 string ...* [standard procedure]
Returns `#t` if all the strings are the same length and contain exactly the same characters in the same positions, otherwise returns `#f`.

`string-ci=?` *string1 string2 string . . .* [char library procedure]
Returns `#t` if, after case-folding, all the strings are the same length and contain the same characters in the same positions, otherwise returns `#f`. Specifically, these procedures behave as if `string-foldcase` were applied to their arguments before comparing them.

`string<?` *string1 string2 string . . .* [standard procedure]
`string-ci<?` *string1 string2 string . . .* [char library procedure]
`string>?` *string1 string2 string . . .* [standard procedure]
`string-ci>?` *string1 string2 string . . .* [char library procedure]
`string<=?` *string1 string2 string . . .* [standard procedure]
`string-ci<=?` *string1 string2 string . . .* [char library procedure]
`string>=?` *string1 string2 string . . .* [standard procedure]
`string-ci>=?` *string1 string2 string . . .* [char library procedure]
These procedures return `#t` if their arguments are (respectively): monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing.

These predicates are required to be transitive.

These procedures compare strings in an implementation-defined way. One approach is to make them the lexicographic extensions to strings of the corresponding orderings on characters. In that case, `string<?` would be the lexicographic ordering on strings induced by the ordering `char<?` on characters, and if the two strings differ in length but are the same up to the length of the shorter string, the shorter string would be considered to be lexicographically less than the longer string. However, it is also permitted to use the natural ordering imposed by the implementation's internal representation of strings, or a more complex locale-specific ordering.

In all cases, a pair of strings must satisfy exactly one of `string<?`, `string=?`, and `string>?`, and must satisfy `string<=?` if and only if they do not satisfy `string>?` and `string>=?` if and only if they do not satisfy `string<?`.

The '`-ci`' procedures behave as if they applied `string-foldcase` to their arguments before invoking the corresponding procedures without '`-ci`'.

`string-compare` *string1 string2 if-eq if-lt if-gt* [procedure]
`string-compare-ci` *string1 string2 if-eq if-lt if-gt* [procedure]
If-eq, *if-lt*, and *if-gt* are procedures of no arguments (thunks). The two strings are compared; if they are equal, *if-eq* is applied, if *string1* is less than *string2*, *if-lt* is applied, else if *string1* is greater than *string2*, *if-gt* is applied. The value of the procedure is the value of the thunk that is applied.

`string-compare` distinguishes uppercase and lowercase letters;
`string-compare-ci` does not.

```
(define (cheer) (display "Hooray!"))
(define (boo)   (display "Boo-hiss!"))
(string-compare "a" "b"  cheer  (lambda() 'ignore)  boo)
        ⊣  Hooray!
        ⇒  unspecified
```

`string-upcase` *string* [char library procedure]
`string-downcase` *string* [char library procedure]
`string-titlecase` *string* [procedure]
`string-foldcase` *string* [char library procedure]

These procedures apply the Unicode full string uppercasing, lowercasing, titlecasing, and case-folding algorithms to their arguments and return the result. In certain cases, the result differs in length from the argument. If the result is equal to the argument in the sense of `string=?`, the argument may be returned. Note that language-sensitive mappings and foldings are not used.

The Unicode Standard prescribes special treatment of the Greek letter Σ, whose normal lower-case form is σ but which becomes ς at the end of a word. See UAX #44 (part of the Unicode Standard) for details. However, implementations of `string-downcase` are not required to provide this behavior, and may choose to change Σ to σ in all cases.

`string-upper-case?` *string* [procedure]
`string-lower-case?` *string* [procedure]

These procedures return `#t` if all the letters in the string are lower case or upper case, otherwise they return `#f`. The string must contain at least one letter or the procedures return `#f`.

```
(map string-upper-case? '(""    "A"    "art"  "Art"  "ART"))
                        ⇒ (#f    #t     #f     #f     #t)
```

`substring` *string* [*start* [*end*]] [standard procedure]

Returns an immutable copy of the part of the given *string* between *start* and *end*.

`string-slice` *string* [*start* [*end*]] [procedure]

Returns a *slice* of *string*, restricted to the range of characters specified by *start* and *end*. The returned slice will be mutable if `string` is mutable, or immutable if `string` is immutable.

A slice is a kind of string that provides a view into another string. The slice behaves like any other string, but changes to a mutable slice are reflected in the original string and vice versa.

```
(define foo (string-copy "abcde"))
foo ⇒ "abcde"

(define bar (string-slice foo 1 4))
bar ⇒ "bcd"

(string-set! foo 2 #\z)
foo ⇒ "abzde"
bar ⇒ "bzd"

(string-set! bar 1 #\y)
bar ⇒ "byd"
foo ⇒ "abyde"
```

`string-append` *string* ... [standard procedure]
`string-append*` *strings* [procedure]
Returns an immutable string whose characters are the concatenation of the characters in the given strings.

The non-standard procedure `string-append*` is identical to `string-append` but takes a single argument that's a list of strings, rather than multiple string arguments.

`string->list` *string* [*start* [*end*]] [standard procedure]
`list->string` *list* [standard procedure]
It is an error if any element of *list* is not a character.

The `string->list` procedure returns a newly allocated list of the characters of *string* between *start* and *end*. `list->string` returns an immutable string formed from the elements in the list *list*. In both procedures, order is preserved. `string->list` and `list->string` are inverses so far as `equal?` is concerned.

`string-copy` *string* [*start* [*end*]] [standard procedure]
Returns a newly allocated mutable copy of the part of the given *string* between *start* and *end*.

`string-copy!` *to at from* [*start* [*end*]] [standard procedure]
It is an error if *to* is not a mutable string or if *at* is less than zero or greater than the length of *to*. It is also an error if `(- (string-length` *to*`)` *at*`)` is less than `(-` *end start*`)`.

Copies the characters of string *from* between *start* and *end* to string *to*, starting at *at*. The order in which characters are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

```
(define a "12345")
(define b (string-copy "abcde"))
(string-copy! b 1 a 0 2) ⇒ 3
b ⇒ "a12de"%
```

Implementation note: in MIT/GNU Scheme `string-copy!` returns the value `(+` *at* `(-` *end start*`))`.

`string-fill!` *string fill* [*start* [*end*]] [standard procedure]
It is an error if *string* is not a mutable string or if *fill* is not a character.

The `string-fill!` procedure stores *fill* in the elements of *string* between *start* and *end*.

The next two procedures treat a given string as a sequence of *grapheme clusters*, a concept defined by the Unicode standard in UAX #29:

> It is important to recognize that what the user thinks of as a "character"—a basic unit of a writing system for a language—may not be just a single Unicode code point. Instead, that basic unit may be made up of multiple Unicode code points. To avoid ambiguity with the computer use of the term character, this is called a user-perceived character. For example, "G" + acute-accent is a

user-perceived character: users think of it as a single character, yet is actually represented by two Unicode code points. These user-perceived characters are approximated by what is called a grapheme cluster, which can be determined programmatically.

`grapheme-cluster-length` *string* [procedure]
This procedure returns the number of grapheme clusters in *string*.

For ASCII strings, this is identical to `string-length`.

`grapheme-cluster-slice` *string start end* [procedure]
This procedure slices *string* at the grapheme-cluster boundaries specified by the *start* and *end* indices. These indices are grapheme-cluster indices, *not* normal string indices.

For ASCII strings, this is identical to `string-slice`.

`string-word-breaks` *string* [procedure]
This procedure returns a list of *word break* indices for *string*, ordered from smallest index to largest. Word breaks are defined by the Unicode standard in UAX #29, and generally coincide with what we think of as the boundaries of words in written text.

MIT/GNU Scheme supports the Unicode canonical normalization forms NFC (*Normalization Form C*) and NFD (*Normalization Form D*). The reason for these forms is that there can be multiple different Unicode sequences for a given text; these sequences are semantically identical and should be treated equivalently for all purposes. If two such sequences are normalized to the same form, the resulting normalized sequences will be identical.

By default, most procedures that return strings do not normalize, so it is up to the programmer to normalize as needed.

Generally speaking, NFC is preferred for most purposes, as it is the minimal-length sequence for the variants. Consult the Unicode standard for the details and for information about why one normalization form is preferable for a specific purpose.

When doing search and match operations, it is recommended that the argument strings be in NFC. Without normalization, strings that should match may not, if they have inconsistent encodings for one or more characters.

`string-in-nfc?` *string* [procedure]
`string-in-nfd?` *string* [procedure]
These procedures return `#t` if *string* is in Unicode Normalization Form C or D respectively. Otherwise they return `#f`.

Note that if *string* consists only of code points strictly less than `#xC0`, then `string-in-nfd?` returns `#t`. If *string* consists only of code points strictly less than `#x300`, then `string-in-nfc?` returns `#t`. Consequently both of these procedures will return `#t` for an ASCII string argument.

`string->nfc` *string* [procedure]
`string->nfd` *string* [procedure]
The procedures convert *string* into Unicode Normalization Form C or D respectively. If *string* is already in the correct form, they return *string* itself, or an immutable copy if *string* is mutable.

`string-map` *proc string string . . .* [standard procedure]

It is an error if *proc* does not accept as many arguments as there are *string*s and return a single character.

The `string-map` procedure applies *proc* element-wise to the elements of the *string*s and returns an immutable string of the results, in order. If more than one *string* is given and not all strings have the same length, `string-map` terminates when the shortest string runs out. The dynamic order in which *proc* is applied to the elements of the *string*s is unspecified. If multiple returns occur from `string-map`, the values returned by earlier returns are not mutated.

```
(string-map char-foldcase "AbdEgH")  ⇒  "abdegh"

(string-map
 (lambda (c)
   (integer->char (+ 1 (char->integer c))))
 "HAL")                  ⇒  "IBM"

(string-map
 (lambda (c k)
   ((if (eqv? k #\u) char-upcase char-downcase) c))
 "studlycaps xxx"
 "ululululul")           ⇒  "StUdLyCaPs"
```

`string-for-each` *proc string string . . .* [standard procedure]

It is an error if *proc* does not accept as many arguments as there are *string*s.

The arguments to `string-for-each` are like the arguments to `string-map`, but `string-for-each` calls *proc* for its side effects rather than for its values. Unlike `string-map`, `string-for-each` is guaranteed to call *proc* on the elements of the *list*s in order from the first element(s) to the last, and the value returned by `string-for-each` is unspecified. If more than one *string* is given and not all strings have the same length, `string-for-each` terminates when the shortest string runs out. It is an error for *proc* to mutate any of the strings.

```
(let ((v '()))
  (string-for-each
   (lambda (c) (set! v (cons (char->integer c) v)))
   "abcde")
  v)                       ⇒  (101 100 99 98 97)
```

`string-count` *proc string string . . .* [procedure]

It is an error if *proc* does not accept as many arguments as there are *string*s.

The `string-count` procedure applies *proc* element-wise to the elements of the *string*s and returns a count of the number of true values it returns. If more than one *string* is given and not all strings have the same length, `string-count` terminates when the shortest string runs out. The dynamic order in which *proc* is applied to the elements of the *string*s is unspecified.

`string-any` *proc string string . . .* [procedure]

It is an error if *proc* does not accept as many arguments as there are *string*s.

The `string-any` procedure applies *proc* element-wise to the elements of the *string*s and returns `#t` if it returns a true value. If *proc* doesn't return a true value, `string-any` returns `#f`.

If more than one *string* is given and not all strings have the same length, `string-any` terminates when the shortest string runs out. The dynamic order in which *proc* is applied to the elements of the *string*s is unspecified.

`string-every` *proc string string ...* [procedure]

It is an error if *proc* does not accept as many arguments as there are *string*s.

The `string-every` procedure applies *proc* element-wise to the elements of the *string*s and returns `#f` if it returns a false value. If *proc* doesn't return a false value, `string-every` returns `#t`.

If more than one *string* is given and not all strings have the same length, `string-every` terminates when the shortest string runs out. The dynamic order in which *proc* is applied to the elements of the *string*s is unspecified.

`string-null?` *string* [procedure]

Returns `#t` if *string* has zero length; otherwise returns `#f`.

```
(string-null? "")       ⇒  #t
(string-null? "Hi")     ⇒  #f
```

`string-hash` *string* [*modulus*] [procedure]
`string-hash-ci` *string* [*modulus*] [procedure]

These SRFI 69 procedures return an exact non-negative integer that can be used for storing the specified *string* in a hash table. Equal strings (in the sense of `string=?` and `string-ci=?` respectively) return equal (`=`) hash codes, and non-equal but similar strings are usually mapped to distinct hash codes.

`string-head` *string end* [procedure]

Equivalent to `(substring` *string* `0` *end*`)`.

`string-tail` *string start* [procedure]

Equivalent to `(substring` *string start*`)`.

If the optional argument *modulus* is specified, it must be an exact positive integer, and the result of the hash computation is restricted to be less than that value. This is equivalent to calling `modulo` on the result, but may be faster.

`string-builder` [*buffer-length*] [procedure]

This procedure returns a *string builder* that can be used to incrementally collect characters and later convert that collection to a string. This is similar to a string output port, but is less general and significantly faster.

The optional *buffer-length* argument, if given, must be an exact positive integer. It controls the size of the internal buffers that are used to accumulate characters. Larger values make the builder somewhat faster but use more space. The default value of this argument is `16`.

The returned string builder is a procedure that accepts zero or one arguments as follows:

- Given a character argument, the string builder appends that character to the string being built and returns an unspecified value.

- Given a string argument, the string builder appends that string to the string being built and returns an unspecified value.
- Given no arguments, or one of the "result" arguments (see below), the string builder returns a copy of the string being built. Note that this does not affect the string being built, so immediately calling the builder with no arguments a second time returns a new copy of the same string.
- Given the argument `empty?`, the string builder returns `#t` if the string being built is empty and `#f` otherwise.
- Given the argument `count`, the string builder returns the size of the string being built.
- Given the argument `reset!`, the string builder discards the string being built and returns to the state it was in when initially created.

The "result" arguments control the form of the returned string. The arguments `immutable` (or no argument) and `mutable` are straightforward, specifying the mutability of the returned string. For these arguments, the returned string contains exactly the same characters, in the same order, as were appended to the builder.

However, calling with the argument `nfc`, returns an immutable string in Unicode Normalization Form C, exactly as if `string->nfc` were called on one of the other two result strings.

`string-joiner` *infix prefix suffix* [procedure]
`string-joiner*` *infix prefix suffix* [procedure]

This procedure's arguments are keyword arguments; that is, each argument is a symbol of the same name followed by its value. The order of the arguments doesn't matter, but each argument may appear only once.

These procedures return a *joiner* procedure that takes multiple strings and joins them together into an immutable string. The joiner returned by `string-joiner` accepts these strings as multiple string arguments, while `string-joiner*` accepts the strings as a single list-valued argument.

The joiner produces a result by adding *prefix* before, *suffix* after, and *infix* between each input string, then concatenating everything together into a single string. Each of the *prefix*, *suffix*, and *infix* arguments is optional and defaults to an empty string, so normally at least one is specified.

Some examples:

```
((string-joiner) "a" "b" "c")
  ⇒  "abc"

((string-joiner 'infix " ") "a" "b" "c")
  ⇒  "a b c"

((string-joiner 'infix ", ") "a" "b" "c")
  ⇒  "a, b, c"

((string-joiner* 'infix ", " 'prefix "<" 'suffix ">")
 '("a" "b" "c"))
```

```
⇒  "<a>, <b>, <c>"
```

`string-splitter` *delimiter allow-runs? copier copy?* [procedure]

This procedure's arguments are keyword arguments; that is, each argument is a symbol of the same name followed by its value. The order of the arguments doesn't matter, but each argument may appear only once.

This procedure returns a *splitter* procedure that splits a given string into parts, returning a list of the parts. This is done by identifying delimiter characters and breaking the string at those delimiters. The splitting process is controlled by the arguments:

- *delimiter* is either a character, a character set, or more generally a procedure that accepts a single character argument and returns a boolean value. The splitter uses this to identify delimiters in the string. The default value of this argument is `char-whitespace?`.
- *allow-runs?* is a boolean that controls what happens when two or more adjacent delimiters are found. If *allow-runs?* is `#t`, then all of the adjacent delimiters are treated as if they were a single delimiter, and the string is split at the beginning and end of the delimiters. If *allow-runs?* is `#f`, then adjacent delimiters are treated as if they were separate with an empty string between them. The default value of this argument is `#t`.
- *copier* is a procedure that accepts three arguments: a string, a start index, and an end index, returning the specified substring as a string. It defaults to `string-slice`.
- *copy?* is a boolean, for backwards compatibility; instead use *copier*. A value of `#t` is equivalent to a *copier* of `substring`, while a value of `#f` is equivalent to a *copier* of `string-slice`.

Some examples:

```
((string-splitter) "a b c")
  ⇒  ("a" "b" "c")

((string-splitter) "a\tb\tc")
  ⇒  ("a" "b" "c")

((string-splitter 'delimiter #\space) "a\tb\tc")
  ⇒  ("a\tb\tc")

((string-splitter) " a  b  c ")
  ⇒  ("a" "b" "c")

((string-splitter 'allow-runs? #f) " a  b  c ")
  ⇒  ("" "a" "" "b" "" "c" "")
```

`string-padder` *where fill-with clip?* [procedure]

This procedure's arguments are keyword arguments; that is, each argument is a symbol of the same name followed by its value. The order of the arguments doesn't matter, but each argument may appear only once.

This procedure returns a *padder* procedure that takes a string and a grapheme-cluster length as its arguments and returns a new string that has been padded to that length. The padder adds grapheme clusters to the string until it has the specified length. If the string's grapheme-cluster length is greater than the given length, the string may, depending on the arguments, be reduced to the specified length.

The padding process is controlled by the arguments:

- *where* is a symbol: either `leading` or `trailing`, which directs the padder to add/remove leading or trailing grapheme clusters. The default value of this argument is `leading`.
- *fill-with* is a string that contains exactly one grapheme cluster, which is used as the padding to increase the size of the string. The default value of this argument is `" "` (a single space character).
- *clip?* is a boolean that controls what happens if the given string has a longer grapheme-cluster length than the given length. If `clip?` is `#t`, grapheme clusters are removed (by slicing) from the string until it is the correct length; if it is `#f` then the string is returned unchanged. The grapheme clusters are removed from the beginning of the string if `where` is `leading`, otherwise from the end of the string. The default value of this argument is `#t`.

Some examples:

```
((string-padder) "abc def" 10)
  ⇒  "   abc def"

((string-padder 'where 'trailing) "abc def" 10)
  ⇒  "abc def   "

((string-padder 'fill-with "X") "abc def" 10)
  ⇒  "XXXabc def"

((string-padder) "abc def" 5)
  ⇒  "c def"

((string-padder 'where 'trailing) "abc def" 5)
  ⇒  "abc d"

((string-padder 'clip? #f) "abc def" 5)
  ⇒  "abc def"
```

`string-pad-left` *string k* [*char*] [obsolete procedure]
`string-pad-right` *string k* [*char*] [obsolete procedure]

These procedures are **deprecated** and should be replaced by use of `string-padder` which is more flexible.

These procedures return an immutable string created by padding *string* out to length *k*, using *char*. If *char* is not given, it defaults to `#\space`. If *k* is less than the length of *string*, the resulting string is a truncated form of *string*. `string-pad-left` adds padding characters or truncates from the beginning of the string (lowest indices), while `string-pad-right` does so at the end of the string (highest indices).

```
(string-pad-left "hello" 4)             ⇒  "ello"
(string-pad-left "hello" 8)             ⇒  "   hello"
(string-pad-left "hello" 8 #\*)         ⇒  "***hello"
(string-pad-right "hello" 4)            ⇒  "hell"
(string-pad-right "hello" 8)            ⇒  "hello   "
```

`string-trimmer` *where to-trim copier copy?* [procedure]

This procedure's arguments are keyword arguments; that is, each argument is a symbol of the same name followed by its value. The order of the arguments doesn't matter, but each argument may appear only once.

This procedure returns a *trimmer* procedure that takes a string as its argument and trims that string, returning the trimmed result. The trimming process is controlled by the arguments:

- *where* is a symbol: either `leading`, `trailing`, or `both`, which directs the trimmer to trim leading characters, trailing characters, or both. The default value of this argument is `both`.
- *to-trim* is either a character, a character set, or more generally a procedure that accepts a single character argument and returns a boolean value. The trimmer uses this to identify characters to remove. The default value of this argument is `char-whitespace?`.
- *copier* is a procedure that accepts three arguments: a string, a start index, and an end index, returning the specified substring as a string. It defaults to `string-slice`.
- *copy?* is a boolean, for backwards compatibility; instead use *copier*. A value of `#t` is equivalent to a *copier* of `substring`, while a value of `#f` is equivalent to a *copier* of `string-slice`.

Some examples:

```
((string-trimmer 'where 'leading) "    ABC   DEF    ")
  ⇒  "ABC   DEF    "

((string-trimmer 'where 'trailing) "    ABC   DEF    ")
  ⇒  "    ABC   DEF"

((string-trimmer 'where 'both) "    ABC   DEF    ")
  ⇒  "ABC   DEF"

((string-trimmer) "    ABC   DEF    ")
  ⇒  "ABC   DEF"

((string-trimmer 'to-trim char-numeric? 'where 'leading)
 "21 East 21st Street #3")
  ⇒  " East 21st Street #3"

((string-trimmer 'to-trim char-numeric? 'where 'trailing)
 "21 East 21st Street #3")
```

```
  ⇒  "21 East 21st Street #"

((string-trimmer 'to-trim char-numeric?)
 "21 East 21st Street #3")
  ⇒  " East 21st Street #"
```

`string-trim` *string* [*char-set*] [obsolete procedure]
`string-trim-left` *string* [*char-set*] [obsolete procedure]
`string-trim-right` *string* [*char-set*] [obsolete procedure]

These procedures are **deprecated** and should be replaced by use of `string-trimmer` which is more flexible.

Returns an immutable string created by removing all characters that are not in *char-set* from: (`string-trim`) both ends of *string*; (`string-trim-left`) the beginning of *string*; or (`string-trim-right`) the end of *string*. *Char-set* defaults to `char-set:not-whitespace`.

```
(string-trim "  in the end  ")          ⇒  "in the end"
(string-trim "              ")          ⇒  ""
(string-trim "100th" char-set:numeric)  ⇒  "100"
(string-trim-left "-.-+-=-" (char-set #\+))
                                        ⇒  "+-=-"
(string-trim "but (+ x y) is" (char-set #\( #\)))
                                        ⇒  "(+ x y)"
```

`string-replace` *string char1 char2* [procedure]

Returns an immutable string containing the same characters as *string* except that all instances of *char1* have been replaced by *char2*.

6.1 Searching and Matching Strings

This section describes procedures for searching a string, either for a character or a substring, and matching two strings to one another.

`string-search-forward` *pattern string* [*start* [*end*]] [procedure]

Searches *string* for the leftmost occurrence of the substring *pattern*. If successful, the index of the first character of the matched substring is returned; otherwise, `#f` is returned.

```
(string-search-forward "rat" "pirate")
    ⇒ 2
(string-search-forward "rat" "pirate rating")
    ⇒ 2
(string-search-forward "rat" "pirate rating" 4 13)
    ⇒ 7
(string-search-forward "rat" "pirate rating" 9 13)
    ⇒ #f
```

`string-search-backward` *pattern string* [*start* [*end*]] [procedure]

Searches *string* for the rightmost occurrence of the substring *pattern*. If successful, the index to the right of the last character of the matched substring is returned; otherwise, `#f` is returned.

```
(string-search-backward "rat" "pirate")
    ⇒ 5
(string-search-backward "rat" "pirate rating")
    ⇒ 10
(string-search-backward "rat" "pirate rating" 1 8)
    ⇒ 5
(string-search-backward "rat" "pirate rating" 9 13)
    ⇒ #f
```

`string-search-all` *pattern string* [*start* [*end*]] [procedure]

Searches *string* to find all occurrences of the substring *pattern*. Returns a list of the occurrences; each element of the list is an index pointing to the first character of an occurrence.

```
(string-search-all "rat" "pirate")
    ⇒ (2)
(string-search-all "rat" "pirate rating")
    ⇒ (2 7)
(string-search-all "rat" "pirate rating" 4 13)
    ⇒ (7)
(string-search-all "rat" "pirate rating" 9 13)
    ⇒ ()
```

`substring?` *pattern string* [procedure]

Searches *string* to see if it contains the substring *pattern*. Returns `#t` if *pattern* is a substring of *string*, otherwise returns `#f`.

```
(substring? "rat" "pirate")             ⇒  #t
(substring? "rat" "outrage")            ⇒  #f
(substring? "" any-string)              ⇒  #t
(if (substring? "moon" text)
    (process-lunar text)
    'no-moon)
```

`string-find-first-index` *proc string string* ... [procedure]
`string-find-last-index` *proc string string* ... [procedure]

The *proc* argument must accept as many arguments as there are *string*s.

These procedures apply *proc* element-wise to the elements of the *string*s and return the first or last index for which *proc* returns a true value. If there is no such index, then `#f` is returned.

If more than one *string* is given and not all strings have the same length, then only the indexes of the shortest string are tested.

`string-find-next-char` *string char* [*start* [*end*]] [procedure]
`string-find-next-char-ci` *string char* [*start* [*end*]] [procedure]
`string-find-next-char-in-set` *string char-set* [*start* [*end*]] [procedure]

These procedures search *string* for a matching character, starting from *start* and moving forwards to *end*. If there is a matching character, the procedures stop the search and return the index of that character. If there is no matching character, the procedures return `#f`.

The procedures differ only in how they match characters: `string-find-next-char` matches a character that is `char=?` to *char*; `string-find-next-char-ci` matches a character that is `char-ci=?` to *char*; and `string-find-next-char-in-set` matches a character that's a member of *char-set*.

```
(string-find-next-char "Adam" #\A)           ⇒  0
(string-find-next-char "Adam" #\A 1 4)       ⇒  #f
(string-find-next-char-ci "Adam" #\A 1 4)    ⇒  2
(string-find-next-char-in-set my-string char-set:alphabetic)
    ⇒  start position of the first word in my-string
; Can be used as a predicate:
(if (string-find-next-char-in-set my-string
                                  (char-set #\( #\) ))
    'contains-parentheses
    'no-parentheses)
```

`string-find-previous-char` *string char* [*start* [*end*]] [procedure]
`string-find-previous-char-ci` *string char* [*start* [*end*]] [procedure]
`string-find-previous-char-in-set` *string char-set* [*start* [*end*]] [procedure]

These procedures search *string* for a matching character, starting from *end* and moving backwards to *start*. If there is a matching character, the procedures stop the search and return the index of that character. If there is no matching character, the procedures return `#f`.

The procedures differ only in how they match characters: `string-find-previous-char` matches a character that is `char=?` to *char*; `string-find-previous-char-ci` matches a character that is `char-ci=?` to *char*; and `string-find-previous-char-in-set` matches a character that's a member of *char-set*.

`string-match-forward` *string1 string2* [procedure]

Compares the two strings, starting from the beginning, and returns the number of characters that are the same. If the two strings start differently, returns 0.

```
(string-match-forward "mirror" "micro") ⇒  2  ; matches "mi"
(string-match-forward "a" "b")          ⇒  0  ; no match
```

`string-match-backward` *string1 string2* [procedure]

Compares the two strings, starting from the end and matching toward the front, returning the number of characters that are the same. If the two strings end differently, returns 0.

```
(string-match-backward "bulbous" "fractious")
                                        ⇒  3  ; matches "ous"
```

`string-prefix?` *string1 string2* [procedure]
`string-prefix-ci?` *string1 string2* [procedure]

These procedures return `#t` if the first string forms the prefix of the second; otherwise returns `#f`. The `-ci` procedures don't distinguish uppercase and lowercase letters.

```
(string-prefix? "abc" "abcdef")         ⇒  #t
(string-prefix? "" any-string)          ⇒  #t
```

`string-suffix?` *string1 string2* [procedure]
`string-suffix-ci?` *string1 string2* [procedure]

These procedures return `#t` if the first string forms the suffix of the second; otherwise returns `#f`. The `-ci` procedures don't distinguish uppercase and lowercase letters.

```
(string-suffix? "ous" "bulbous")        ⇒  #t
(string-suffix? "" any-string)          ⇒  #t
```

6.2 Regular Expressions

MIT/GNU Scheme provides support for matching and searching strings against regular expressions. This is considerably more flexible than ordinary string matching and searching, but potentially much slower. On the other hand it is less powerful than the mechanism described in Section 14.14 [Parser Language], page 257.

Traditional regular expressions are defined with string patterns in which characters like '`[`' and '`*`' have special meanings. Unfortunately, the syntax of these patterns is not only baroque but also comes in many different and mutually-incompatible varieties. As a consequence we have chosen to specify regular expressions using an s-expression syntax, which we call a *regular s-expression*, abbreviated as *regsexp*.

Previous releases of MIT/GNU Scheme provided a regular-expression implementation nearly identical to that of GNU Emacs version 18. This implementation supported only 8-bit strings, which made it unsuitable for use with Unicode strings. This implementation still exists but is deprecated and will be removed in a future release.

6.2.1 Regular S-Expressions

A regular s-expression is either a character or a string, which matches itself, or one of the following forms.

Examples in this section use the following definitions for brevity:

```
(define (try-match pattern string)
  (regsexp-match-string (compile-regsexp pattern) string))

(define (try-search pattern string)
  (regsexp-search-string-forward (compile-regsexp pattern) string))
```

These forms match one or more characters literally:

`char-ci` *char* [regsexp]

Matches *char* without considering case.

`string-ci` *string* [regsexp]

Matches *string* without considering case.

`any-char` [regsexp]

Matches one character other than `#\newline`.

```
(try-match '(any-char) "") ⇒ #f
(try-match '(any-char) "a") ⇒ (0 1)
(try-match '(any-char) "\n") ⇒ #f
(try-search '(any-char) "") ⇒ #f
(try-search '(any-char) "ab") ⇒ (0 1)
(try-search '(any-char) "\na") ⇒ (1 2)
```

`char-in` *datum ...* [regsexp]
`char-not-in` *datum ...* [regsexp]
Matches one character in (not in) the character set specified by `(char-set` *datum ...*`)`.

```
(try-match '(seq "a" (char-in "ab") "c") "abc") ⇒ (0 3)
(try-match '(seq "a" (char-not-in "ab") "c") "abc") ⇒ #f
(try-match '(seq "a" (char-not-in "ab") "c") "adc") ⇒ (0 3)
(try-match '(seq "a" (+ (char-in numeric)) "c") "a019c") ⇒ (0 5)
```

These forms match no characters, but only at specific locations in the input string:

`line-start` [regsexp]
`line-end` [regsexp]
Matches no characters at the start (end) of a line.

```
(try-match '(seq (line-start)
                 (* (any-char))
                 (line-end))
           "abc") ⇒ (0 3)
(try-match '(seq (line-start)
                 (* (any-char))
                 (line-end))
           "ab\nc") ⇒ (0 2)
(try-search '(seq (line-start)
                  (* (char-in alphabetic))
                  (line-end))
            "1abc") ⇒ #f
(try-search '(seq (line-start)
                  (* (char-in alphabetic))
                  (line-end))
            "1\nabc") ⇒ (2 5)
```

`string-start` [regsexp]
`string-end` [regsexp]
Matches no characters at the start (end) of the string.

```
(try-match '(seq (string-start)
                 (* (any-char))
                 (string-end))
           "abc") ⇒ (0 3)
(try-match '(seq (string-start)
                 (* (any-char))
                 (string-end))
           "ab\nc") ⇒ #f
(try-search '(seq (string-start)
                  (* (char-in alphabetic))
                  (string-end))
            "1abc") ⇒ #f
```

```
(try-search '(seq (string-start)
                  (* (char-in alphabetic))
                  (string-end))
            "1\nabc") ⇒ #f
```

These forms match repetitions of a given regsexp. Most of them come in two forms, one of which is *greedy* and the other *shy*. The greedy form matches as many repetitions as it can, then uses failure backtracking to reduce the number of repetitions one at a time. The shy form matches the minimum number of repetitions, then uses failure backtracking to increase the number of repetitions one at a time. The shy form is similar to the greedy form except that a `?` is added at the end of the form's keyword.

`?` *regsexp* [regsexp]
`??` *regsexp* [regsexp]

Matches *regsexp* zero or one time.

```
(try-search '(seq (char-in alphabetic)
                  (? (char-in numeric)))
            "a") ⇒ (0 1)
(try-search '(seq (char-in alphabetic)
                  (?? (char-in numeric)))
            "a") ⇒ (0 1)
(try-search '(seq (char-in alphabetic)
                  (? (char-in numeric)))
            "a1") ⇒ (0 2)
(try-search '(seq (char-in alphabetic)
                  (?? (char-in numeric)))
            "a1") ⇒ (0 1)
(try-search '(seq (char-in alphabetic)
                  (? (char-in numeric)))
            "1a2") ⇒ (1 3)
(try-search '(seq (char-in alphabetic)
                  (?? (char-in numeric)))
            "1a2") ⇒ (1 2)
```

`*` *regsexp* [regsexp]
`*?` *regsexp* [regsexp]

Matches *regsexp* zero or more times.

```
(try-match '(seq (char-in alphabetic)
                 (* (char-in numeric))
                 (any-char))
           "aa") ⇒ (0 2)
(try-match '(seq (char-in alphabetic)
                 (*? (char-in numeric))
                 (any-char))
           "aa") ⇒ (0 2)
```

```
(try-match '(seq (char-in alphabetic)
                 (* (char-in numeric))
                 (any-char))
           "a123a") ⇒ (0 5)
(try-match '(seq (char-in alphabetic)
                 (*? (char-in numeric))
                 (any-char))
           "a123a") ⇒ (0 2)
```

`+` *regsexp* [regsexp]
`+?` *regsexp* [regsexp]

Matches *regsexp* one or more times.

```
(try-match '(seq (char-in alphabetic)
                 (+ (char-in numeric))
                 (any-char))
           "aa") ⇒ #f
(try-match '(seq (char-in alphabetic)
                 (+? (char-in numeric))
                 (any-char))
           "aa") ⇒ #f
(try-match '(seq (char-in alphabetic)
                 (+ (char-in numeric))
                 (any-char))
           "a123a") ⇒ (0 5)
(try-match '(seq (char-in alphabetic)
                 (+? (char-in numeric))
                 (any-char))
           "a123a") ⇒ (0 3)
```

`**` *n m regsexp* [regsexp]
`**?` *n m regsexp* [regsexp]

The *n* argument must be an exact nonnegative integer. The *m* argument must be either an exact integer greater than or equal to *n*, or else `#f`.

Matches *regsexp* at least *n* times and at most *m* times; if *m* is `#f` then there is no upper limit.

```
(try-match '(seq (char-in alphabetic)
                 (** 0 2 (char-in numeric))
                 (any-char))
           "aa") ⇒ (0 2)
(try-match '(seq (char-in alphabetic)
                 (**? 0 2 (char-in numeric))
                 (any-char))
           "aa") ⇒ (0 2)
(try-match '(seq (char-in alphabetic)
                 (** 0 2 (char-in numeric))
                 (any-char))
           "a123a") ⇒ (0 4)
```

```
(try-match '(seq (char-in alphabetic)
                 (**? 0 2 (char-in numeric))
                 (any-char))
           "a123a") ⇒ (0 2)
```

`**` *n regsexp* [regsexp]
: This is an abbreviation for `(** n n regsexp)`. This matcher is neither greedy nor shy since it matches a fixed number of repetitions.

These forms implement alternatives and sequencing:

`alt` *regsexp ...* [regsexp]
: Matches one of the *regsexp* arguments, trying each in order from left to right.

```
(try-match '(alt #\a (char-in numeric)) "a") ⇒ (0 1)
(try-match '(alt #\a (char-in numeric)) "b") ⇒ #f
(try-match '(alt #\a (char-in numeric)) "1") ⇒ (0 1)
```

`seq` *regsexp ...* [regsexp]
: Matches the first *regsexp*, then continues the match with the next *regsexp*, and so on until all of the arguments are matched.

```
(try-match '(seq #\a #\b) "a") ⇒ #f
(try-match '(seq #\a #\b) "aa") ⇒ #f
(try-match '(seq #\a #\b) "ab") ⇒ (0 2)
```

These forms implement named *registers*, which store matched segments of the input string:

`group` *key regsexp* [regsexp]
: The *key* argument must be a fixnum, a character, or a symbol.

  Matches *regsexp*. If the match succeeds, the matched segment is stored in the register named *key*.

```
(try-match '(seq (group a (any-char))
                 (group b (any-char))
                 (any-char))
           "radar") ⇒ (0 3 (a . "r") (b . "a"))
```

`group-ref` *key* [regsexp]
: The *key* argument must be a fixnum, a character, or a symbol.

  Matches the characters stored in the register named *key*. It is an error if that register has not been initialized with a corresponding `group` expression.

```
(try-match '(seq (group a (any-char))
                 (group b (any-char))
                 (any-char)
                 (group-ref b)
                 (group-ref a))
           "radar") ⇒ (0 5 (a . "r") (b . "a"))
```

6.2.2 Regsexp Procedures

The regular s-expression implementation has two parts, like many other regular-expression implementations: a compiler that translates the pattern into an efficient form, and one or more procedures that use that pattern to match or search inputs.

`compile-regsexp` *regsexp* [procedure]
Compiles *regsexp* by translating it into a procedure that implements the specified matcher.

The match and search procedures each return a list when they are successful, and `#f` when they fail. The returned list is of the form `(`*`s e register ...`*`)`, where *s* is the index at which the match starts, *e* is the index at which the match ends, and each *register* is a pair `(`*`key . contents`*`)` where *key* is the register's name and *contents* is the contents of that register as a string.

In order to get reliable results, the string arguments to these procedures must be in Unicode Normalization Form C. The string implementation keeps most strings in this form by default; in other cases the caller must convert the string using `string->nfc`.

`regsexp-match-string` *crse string* [*start* [*end*]] [procedure]
The *crse* argument must be a value returned by `compile-regsexp`.

Matches *string* against *crse* and returns the result.

`regsexp-search-string-forward` *crse string* [*start* [*end*]] [procedure]
The *crse* argument must be a value returned by `compile-regsexp`.

Searches *string* from left to right for a match against *crse* and returns the result.

7 Lists

A *pair* (sometimes called a *dotted pair*) is a data structure with two fields called the *car* and *cdr* fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. The car and cdr fields are assigned by the procedures `set-car!` and `set-cdr!`.

Pairs are used primarily to represent *lists*. A list can be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set X such that

- The empty list is in X.
- If *list* is in X, then any pair whose cdr field contains *list* is also in X.

The objects in the car fields of successive pairs of a list are the *elements* of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The *length* of a list is the number of elements, which is the same as the number of pairs. The *empty list* is a special object of its own type (it is not a pair); it has no elements and its length is zero.[1]

The most general notation (external representation) for Scheme pairs is the "dotted" notation `(`*c1* `.` *c2*`)` where *c1* is the value of the car field and *c2* is the value of the cdr field. For example, `(4 . 5)` is a pair whose car is `4` and whose cdr is `5`. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()`. For example, the following are equivalent notations for a list of symbols:

```
(a b c d e)
(a . (b . (c . (d . (e . ()))))))
```

Whether a given pair is a list depends upon what is stored in the cdr field. When the `set-cdr!` procedure is used, an object can be a list one moment and not the next:

```
(define x (list 'a 'b 'c))
(define y x)
y                                   ⇒ (a b c)
(list? y)                           ⇒ #t
(set-cdr! x 4)                      ⇒ unspecified
x                                   ⇒ (a . 4)
(eqv? x y)                          ⇒ #t
y                                   ⇒ (a . 4)
(list? y)                           ⇒ #f
(set-cdr! x x)                      ⇒ unspecified
(list? y)                           ⇒ #f
```

A chain of pairs that doesn't end in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists, as the following equivalent notations show:

```
(a b c . d)
(a . (b . (c . d)))
```

[1] The above definitions imply that all lists have finite length and are terminated by the empty list.

Within literal expressions and representations of objects read by the `read` procedure, the forms `'`*datum*, `` `*datum* ``, `,`*datum*, and `,@`*datum* denote two-element lists whose first elements are the symbols `quote`, `quasiquote`, `unquote`, and `unquote-splicing`, respectively. The second element in each case is *datum*. This convention is supported so that arbitrary Scheme programs may be represented as lists. Among other things, this permits the use of the `read` procedure to parse Scheme programs.

7.1 Pairs

This section describes the simple operations that are available for constructing and manipulating arbitrary graphs constructed from pairs.

`pair?` *object* [standard procedure]

Returns `#t` if *object* is a pair; otherwise returns `#f`.

```
(pair? '(a . b))                    ⇒ #t
(pair? '(a b c))                    ⇒ #t
(pair? '())                         ⇒ #f
(pair? '#(a b))                     ⇒ #f
```

`cons` *obj1 obj2* [standard procedure]

Returns a newly allocated pair whose car is *obj1* and whose cdr is *obj2*. The pair is guaranteed to be different (in the sense of `eqv?`) from every previously existing object.

```
(cons 'a '())                       ⇒ (a)
(cons '(a) '(b c d))                ⇒ ((a) b c d)
(cons "a" '(b c))                   ⇒ ("a" b c)
(cons 'a 3)                         ⇒ (a . 3)
(cons '(a b) 'c)                    ⇒ ((a b) . c)
```

`xcons` *obj1 obj2* [SRFI 1 procedure]

Returns a newly allocated pair whose car is *obj2* and whose cdr is *obj1*.

```
(xcons '(b c) 'a)                   ⇒ (a b c)
```

`car` *pair* [standard procedure]

Returns the contents of the car field of *pair*. Note that it is an error to take the `car` of the empty list.

```
(car '(a b c))                      ⇒ a
(car '((a) b c d))                  ⇒ (a)
(car '(1 . 2))                      ⇒ 1
(car '())                           error  Illegal datum
```

`cdr` *pair* [standard procedure]

Returns the contents of the cdr field of *pair*. Note that it is an error to take the `cdr` of the empty list.

```
(cdr '((a) b c d))                  ⇒ (b c d)
(cdr '(1 . 2))                      ⇒ 2
(cdr '())                           error  Illegal datum
```

`car+cdr` *pair* [SRFI 1 procedure]

The fundamental pair deconstructor:

```
(lambda (p) (values (car p) (cdr p)))
```

```
(receive (a b) (car+cdr (cons 1 2))
  (write-line a)
  (write-line b))
⊣ 1
⊣ 2
```

`set-car!` *pair object* [standard procedure]

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

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)                    ⇒ unspecified
(set-car! (g) 3)                    error  Illegal datum
```

`set-cdr!` *pair object* [standard procedure]

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

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

`cddddr` *pair* [standard procedure]

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

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

`general-car-cdr` *object path* [procedure]

This procedure is a generalization of `car` and `cdr`. *Path* encodes a particular sequence of `car` and `cdr` operations, which `general-car-cdr` executes on *object*. *Path* is an exact non-negative integer that encodes the operations in a bitwise fashion: a zero bit represents a `cdr` operation, and a one bit represents a `car`. The bits are executed LSB to MSB, and the most significant one bit, rather than being interpreted as an operation, signals the end of the sequence.[2]

For example, the following are equivalent:

```
(general-car-cdr object #b1011)
(cdr (car (car object)))
```

Here is a partial table of path/operation equivalents:

```
#b10    cdr
#b11    car
#b100   cddr
#b101   cdar
#b110   cadr
#b111   caar
#b1000  cdddr
```

`tree-copy` *tree* [SRFI 1 procedure]

This copies an arbitrary *tree* constructed from pairs, copying both the car and cdr elements of every pair. This could have been defined by

```
(define (tree-copy tree)
  (let loop ((tree tree))
    (if (pair? tree)
        (cons (loop (car tree)) (loop (cdr tree)))
        tree)))
```

7.2 Construction of Lists

`list` *object ...* [standard procedure]

Returns a list of its arguments.

```
(list 'a (+ 3 4) 'c)                    ⇒ (a 7 c)
(list)                                  ⇒ ()
```

These expressions are equivalent:

```
(list obj1 obj2 ... objN)
(cons obj1 (cons obj2 ... (cons objN '()) ...))
```

[2] Note that *path* is restricted to a machine-dependent range, usually the size of a machine word. On many machines, this means that the maximum length of *path* will be 30 operations (32 bits, less the sign bit and the "end-of-sequence" bit).

`make-list` *n* [*fill*] [SRFI 1 procedure]

Returns an *n*-element list, whose elements are all the value *fill*. If the *fill* argument is not given, the elements of the list may be arbitrary values.

```
(make-list 4 'c)                    ⇒ (c c c c)
```

`cons*` *object object ...* [SRFI 1 procedure]

`cons*` is similar to `list`, except that `cons*` conses together the last two arguments rather than consing the last argument with the empty list. If the last argument is not a list the result is an improper list. If the last argument is a list, the result is a list consisting of the initial arguments and all of the items in the final argument. If there is only one argument, the result is the argument.

```
(cons* 'a 'b 'c)                    ⇒ (a b . c)
(cons* 'a 'b '(c d))                ⇒ (a b c d)
(cons* 'a)                          ⇒ a
```

These expressions are equivalent:

```
(cons* obj1 obj2 ... objN-1 objN)
(cons obj1 (cons obj2 ... (cons objN-1 objN) ...))
```

`list-tabulate` *k init-proc* [SRFI 1 procedure]
`make-initialized-list` *k init-proc* [obsolete procedure]

Returns a *k*-element list. Element *i* of the list, where 0 <= *i* < *k*, is produced by (*init-proc i*). No guarantee is made about the dynamic order in which *init-proc* is applied to these indices.

```
(list-tabulate 4 values) => (0 1 2 3)
```

`list-copy` *list* [SRFI 1 procedure]

Returns a newly allocated copy of *list*. This copies each of the pairs comprising *list*. This could have been defined by

```
(define (list-copy list)
  (if (null? list)
      '()
      (cons (car list)
            (list-copy (cdr list)))))
```

`iota` *count* [*start* [*step*]] [SRFI 1 procedure]

Returns a list containing the elements

```
(start start+step ... start+(count-1)*step)
```

Count must be an exact non-negative integer, while *start* and *step* can be any numbers. The *start* and *step* parameters default to 0 and 1, respectively.

```
(iota 5) ⇒ (0 1 2 3 4)
(iota 5 0 -0.1) ⇒ (0 -0.1 -0.2 -0.3 -0.4)
```

`vector->list` *vector* [*start* [*end*]] [standard procedure]
`subvector->list` *vector start end* [obsolete procedure]

Returns a newly allocated list of the elements of *vector* between *start* inclusive and *end* exclusive. The inverse of `vector->list` is `list->vector`.

```
(vector->list '#(dah dah didah))        ⇒ (dah dah didah)
```

7.3 Selecting List Components

`list?` *object* [standard procedure]
`proper-list?` *object* [SRFI 1 procedure]

Returns `#t` if *object* is a proper list, otherwise returns `#f`. By definition, all proper lists have finite length and are terminated by the empty list. If *object* is a circular list, returns `#f`.

Any *object* satisfying this predicate will also satisfy exactly one of `pair?` or `null?`.

```
(list? (list 'a 'b 'c))                ⇒ #t
(list? (cons* 'a 'b 'c))               error
(list? (circular-list 'a 'b 'c))       ⇒ #f
```

`circular-list?` *object* [SRFI 1 procedure]

Returns `#t` if *object* is a circular list, otherwise returns `#f`.

```
(circular-list? (list 'a 'b 'c))       ⇒ #f
(circular-list? (cons* 'a 'b 'c))      ⇒ #f
(circular-list? (circular-list 'a 'b 'c)) ⇒ #t
```

`dotted-list?` *object* [SRFI 1 procedure]

Returns `#t` if *object* is an improper list, otherwise returns `#f`.

```
(dotted-list? (list 'a 'b 'c))         ⇒ #f
(dotted-list? (cons* 'a 'b 'c))        ⇒ #t
(dotted-list? (circular-list 'a 'b 'c)) ⇒ #f
```

`length` *list* [standard procedure]

Returns the length of *list*. Signals an error if *list* isn't a proper list.

```
(length (list 'a 'b 'c))               ⇒ 3
(length (cons* 'a 'b 'c))              error
(length (circular-list 'a 'b 'c))      error
```

`length+` *clist* [SRFI 1 procedure]

Clist must be a proper or circular list. If *clist* is a circular list, returns `#f`, otherwise returns the number of pairs comprising the list (which is the same as the length for a proper list).

```
(length+ (list 'a 'b 'c))              ⇒ 3
(length+ (cons* 'a 'b 'c))             error
(length+ (circular-list 'a 'b 'c))     ⇒ #f
```

`count-pairs` *object* [procedure]

Counts the number of pairs in a list-like *object*. If *object* is a proper list, returns the same value as `length`. If *object* is a dotted list, returns the number of pairs including the last one. If *object* is a circular list, counts the number of pairs up to and including the one with the backwards link. If *object* is any other object, returns `0` as apropriate for an empty dotted list.

```
(count-pairs (list 'a 'b 'c))          ⇒ 3
(count-pairs (cons* 'a 'b 'c))         ⇒ 2
(count-pairs (circular-list 'a 'b 'c)) ⇒ 3
```

`null?` *object* [standard procedure]

Returns `#t` if *object* is the empty list; otherwise returns `#f`.

```
(null? '())                             ⇒ #t
(null? (list 'a 'b 'c))                 ⇒ #f
(null? (cons* 'a 'b 'c))                ⇒ #f
(null? (circular-list 'a 'b 'c))        ⇒ #f
```

`null-list?` *list* [SRFI 1 procedure]

List is a proper or circular list. This procedure returns `#t` if the argument is the empty list `()`, and `#f` if the argument is a pair. It is an error to pass this procedure any other value. This procedure is recommended as the termination condition for list-processing procedures that are not defined on dotted lists.

`list-ref` *list k* [standard procedure]

Returns the *k*th element of *list*, using zero-origin indexing. The *valid indexes* of a list are the exact non-negative integers less than the length of the list. The first element of a list has index `0`, the second has index `1`, and so on.

```
(list-ref '(a b c d) 2)                 ⇒ c
(list-ref '(a b c d)
          (exact (round 1.8)))
     ⇒ c
```

`(list-ref` *list k*`)` is equivalent to `(car (drop` *list k*`))`.

`first` *list* [SRFI 1 procedure]
`second` *list* [SRFI 1 procedure]
`third` *list* [SRFI 1 procedure]
`fourth` *list* [SRFI 1 procedure]
`fifth` *list* [SRFI 1 procedure]
`sixth` *list* [SRFI 1 procedure]
`seventh` *list* [SRFI 1 procedure]
`eighth` *list* [SRFI 1 procedure]
`ninth` *list* [SRFI 1 procedure]
`tenth` *list* [SRFI 1 procedure]

Returns the specified element of *list*. It is an error if *list* is not long enough to contain the specified element (for example, if the argument to `seventh` is a list that contains only six elements).

7.4 Cutting and Pasting Lists

`take` *x i* [SRFI 1 procedure]
`drop` *x i* [SRFI 1 procedure]

`take` returns the first *i* elements of list *x*. `drop` returns all but the first *i* elements of list *x*.

```
(take '(a b c d e)  2) => (a b)
(drop '(a b c d e)  2) => (c d e)
```

x may be any value—a proper, circular, or dotted list:

```
(take '(1 2 3 . d) 2) => (1 2)
```

```
(drop '(1 2 3 . d) 2) => (3 . d)
(take '(1 2 3 . d) 3) => (1 2 3)
(drop '(1 2 3 . d) 3) => d
```

For a legal *i*, `take` and `drop` partition the list in a manner which can be inverted with `append`:

```
(append (take x i) (drop x i)) = x
```

`drop` is exactly equivalent to performing *i* `cdr` operations on *x*; the returned value shares a common tail with *x*. If the argument is a list of non-zero length, `take` is guaranteed to return a freshly-allocated list, even in the case where the entire list is taken, e.g. `(take lis (length lis))`.

`list-head` *x i* [obsolete procedure]
`list-tail` *x i* [standard procedure]

Equivalent to `take` and `drop`, respectively. `list-head` is deprecated and should not be used. `list-tail` is defined by R7RS.

`sublist` *list start end* [procedure]

Start and *end* must be exact integers satisfying

```
0 <= start <= end <= (length list)
```

`sublist` returns a newly allocated list formed from the elements of *list* beginning at index *start* (inclusive) and ending at *end* (exclusive).

`append` *list ...* [standard procedure]

Returns a list consisting of the elements of the first *list* followed by the elements of the other *list*s.

```
(append '(x) '(y))                  ⇒ (x y)
(append '(a) '(b c d))              ⇒ (a b c d)
(append '(a (b)) '((c)))            ⇒ (a (b) (c))
(append)                            ⇒ ()
```

The resulting list is always newly allocated, except that it shares structure with the last *list* argument. The last argument may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))            ⇒ (a b c . d)
(append '() 'a)                     ⇒ a
```

`append!` *list ...* [SRFI 1 procedure]

Returns a list that is the argument *list*s concatenated together. The arguments are changed rather than copied. (Compare this with `append`, which copies arguments rather than destroying them.) For example:

```
(define x (list 'a 'b 'c))
(define y (list 'd 'e 'f))
(define z (list 'g 'h))
(append! x y z)                     ⇒ (a b c d e f g h)
x                                   ⇒ (a b c d e f g h)
y                                   ⇒ (d e f g h)
z                                   ⇒ (g h)
```

`last` *pair* [SRFI 1 procedure]
`last-pair` *pair* [SRFI 1 procedure]
`last` returns the last element of the non-empty, finite list *pair*. `last-pair` returns the last pair in the non-empty, finite list *pair*.

```
(last '(a b c)) => c
(last-pair '(a b c)) => (c)
```

`except-last-pair` *list* [obsolete procedure]
`except-last-pair!` *list* [obsolete procedure]
These procedures are deprecated. Instead use `drop-right` or `drop-right!`, respectively, with a second argument of `1`.

7.5 Filtering Lists

`filter` *predicate list* [SRFI 1 procedure]
Returns a newly allocated copy of *list* containing only the elements satisfying *predicate*. *Predicate* must be a procedure of one argument.

```
(filter odd? '(1 2 3 4 5)) ⇒ (1 3 5)
```

`remove` *predicate list* [SRFI 1 procedure]
Like `filter`, except that the returned list contains only those elements *not* satisfying *predicate*.

```
(remove odd? '(1 2 3 4 5)) ⇒ (2 4)
```

`partition` *predicate list* [SRFI 1 procedure]
Partitions the elements of *list* with *predicate*, and returns two values: the list of in-elements and the list of out-elements. The *list* is not disordered—elements occur in the result lists in the same order as they occur in the argument *list*. The dynamic order in which the various applications of `predicate` are made is not specified. One of the returned lists may share a common tail with the argument *list*.

```
(partition symbol? '(one 2 3 four five 6)) =>
    (one four five)
    (2 3 6)
```

`filter!` *predicate list* [SRFI 1 procedure]
`remove!` *predicate list* [SRFI 1 procedure]
`partition!` *predicate list* [SRFI 1 procedure]
Linear-update variants of `filter`, `remove` and `partition`. These procedures are allowed, but not required, to alter the cons cells in the argument *list* to construct the result lists.

`delete` *x list* [*compare*] [SRFI 1 procedure]
`delete!` *x list* [*compare*] [SRFI 1 procedure]
`delete` uses the comparison procedure *compare*, which defaults to `equal?`, to find all elements of *list* that are equal to *x*, and deletes them from *list*. The dynamic order in which the various applications of *compare* are made is not specified.

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 result may share a common tail with the argument list.

Note that fully general element deletion can be performed with the `remove` and `remove!` procedures, e.g.:

```
;; Delete all the even elements from LIS:
(remove even? lis)
```

The comparison procedure is used in this way: (*compare* *x* e_i). That is, *x* is always the first argument, and a list element is always the second argument. The comparison procedure will be used to compare each element of *list* exactly once; the order in which it is applied to the various e_i is not specified. Thus, one can reliably remove all the numbers greater than five from a list with `(delete 5 list <)`.

`delete!` is the linear-update variant of `delete`. It is allowed, but not required, to alter the cons cells in its argument list to construct the result.

`delq` *x list* [procedure]
`delq!` *x list* [procedure]
`delv` *x list* [procedure]
`delv!` *x list* [procedure]

Equivalent to `(delete` *x list* `eq?)`, `(delete!` *x list* `eq?)`, `(delete` *x list* `eqv?)`, and `(delete!` *x list* `eqv?)`, respectively.

`delete-member-procedure` *deletor predicate* [procedure]

Returns a deletion procedure similar to `delv` or `delete!`. *Deletor* should be one of the procedures `list-deletor` or `list-deletor!`. *Predicate* must be an equivalence predicate. The returned procedure accepts exactly two arguments: first, an object to be deleted, and second, a list of objects from which it is to be deleted. If *deletor* is `list-deletor`, the procedure returns a newly allocated copy of the given list in which all entries equal to the given object have been removed. If *deletor* is `list-deletor!`, the procedure returns a list consisting of the top-level elements of the given list with all entries equal to the given object removed; the given list is destructively modified to produce the result. In either case *predicate* is used to compare the given object to the elements of the given list.

Here are some examples that demonstrate how `delete-member-procedure` could have been used to implement `delv` and `delete!`:

```
(define delv
  (delete-member-procedure list-deletor eqv?))
(define delete!
  (delete-member-procedure list-deletor! equal?))
```

`list-deletor` *predicate* [procedure]
`list-deletor!` *predicate* [procedure]

These procedures each return a procedure that deletes elements from lists. *Predicate* must be a procedure of one argument. The returned procedure accepts exactly one argument, which must be a proper list, and applies *predicate* to each of the elements of the argument, deleting those for which it is true.

The procedure returned by `list-deletor` deletes elements non-destructively, by returning a newly allocated copy of the argument with the appropriate elements removed. The procedure returned by `list-deletor!` performs a destructive deletion.

7.6 Searching Lists

`find` *predicate list* [SRFI 1 procedure]

Returns the first element in *list* for which *predicate* is true; returns `#f` if it doesn't find such an element. *Predicate* must be a procedure of one argument.

```
(find even? '(3 1 4 1 5 9)) => 4
```

Note that `find` has an ambiguity in its lookup semantics—if `find` returns `#f`, you cannot tell (in general) if it found a `#f` element that satisfied *predicate*, or if it did not find any element at all. In many situations, this ambiguity cannot arise—either the list being searched is known not to contain any `#f` elements, or the list is guaranteed to have an element satisfying *predicate*. However, in cases where this ambiguity can arise, you should use `find-tail` instead of `find`—`find-tail` has no such ambiguity:

```
(cond ((find-tail pred lis)
       => (lambda (pair) ...)) ; Handle (CAR PAIR)
      (else ...)) ; Search failed.
```

`find-tail` *predicate list* [SRFI 1 procedure]

Returns the first pair of *list* whose car satisfies *predicate*; returns `#f` if there's no such pair. `find-tail` can be viewed as a general-predicate variant of *memv*.

`memq` *object list* [standard procedure]
`memv` *object list* [standard procedure]
`member` *object list* [*compare*] [standard procedure]

These procedures return the first pair of *list* whose car is *object*; the returned pair is always one from which *list* is composed. If *object* does not occur in *list*, `#f` (n.b.: not the empty list) is returned. `memq` uses `eq?` to compare *object* with the elements of *list*, while `memv` uses `eqv?` and `member` uses *compare*, or `equal?` if *compare* is not supplied.[3]

```
(memq 'a '(a b c))                  ⇒ (a b c)
(memq 'b '(a b c))                  ⇒ (b c)
(memq 'a '(b c d))                  ⇒ #f
(memq (list 'a) '(b (a) c))         ⇒ #f
(member (list 'a) '(b (a) c))       ⇒ ((a) c)
(memq 101 '(100 101 102))           ⇒ unspecified
(memv 101 '(100 101 102))           ⇒ (101 102)
```

`member-procedure` *predicate* [procedure]

Returns a procedure similar to `memq`, except that *predicate*, which must be an equivalence predicate, is used instead of `eq?`. This could be used to define `memv` as follows:

```
(define memv (member-procedure eqv?))
```

[3] Although they are often used as predicates, `memq`, `memv`, and `member` do not have question marks in their names because they return useful values rather than just `#t` or `#f`.

7.7 Mapping of Lists

`map` *procedure list list ...* [standard procedure]
It is an error if *procedure* does not accept as many arguments as there are *list*s and return a single value.

The `map` procedure applies *procedure* element-wise to the elements of the *list*s and returns a list of the results, in order. If more than one *list* is given and not all lists are the same length, `map` terminates when the shortest list runs out. The *list*s can be circular, but it is an error if all of them are circular. It is an error for *procedure* to mutate any of the lists. The dynamic order in which *procedure* is applied to the elements of the *list*s is unspecified. If multiple returns occur from `map`, the values returned by earlier returns are not mutated.

```
(map cadr '((a b) (d e) (g h)))         ⇒ (b e h)
(map (lambda (n) (expt n n)) '(1 2 3 4 5))
                                        ⇒ (1 4 27 256 3125)
(map + '(1 2 3) '(4 5 6 7))             ⇒ (5 7 9)
(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))                         ⇒ (1 2) or (2 1)
```

`map*` *knil proc list₁ list₂ ...* [obsolete procedure]
Deprecated, use `fold-right` instead. Equivalent to

```
(fold-right (lambda (e₁ e₂ ... acc)
              (cons* (proc e₁)
                     (proc e₂)
                     ...
                     acc))
            knil
            list₁
            list₂
            ...)
```

`append-map` *procedure list list ...* [SRFI 1 procedure]
Similar to `map` except that the results of applying *procedure* to the elements of *list*s are concatenated together by `append` rather than by `cons`. The following are equivalent, except that the former is more efficient:

```
(append-map procedure list₁ list₂ ...)
(apply append (map procedure list₁ list₂ ...))
```

`append-map*` *knil proc list₁ list₂ ...* [obsolete procedure]
Deprecated, use `fold-right` instead. Equivalent to

```
(fold-right (lambda (e1 e2 ... acc)
              (append (proc e1)
                      (proc e2)
                      ...
                      acc))
            knil
            list1
            list2
            ...)
```

`append-map!` *proc list list ...* [SRFI 1 procedure]

Similar to `map` except that the results of applying *proc* to the elements of *list*s are concatenated together by `append!` rather than by `cons`. The following are equivalent, except that the former is more efficient:

```
(append-map! proc list list ...)
(apply append! (map proc list list ...))
```

`append-map*!` *knil proc list*$_1$ *list*$_2$ *...* [obsolete procedure]

Deprecated, use `fold-right` instead. Equivalent to

```
(fold-right (lambda (e1 e2 ... acc)
              (append! (proc e1)
                       (proc e2)
                       ...
                       acc))
            knil
            list1
            list2
            ...)
```

`for-each` *procedure list list ...* [standard procedure]

It is an error if *procedure* does not accept as many arguments as there are *list*s.

The arguments to `for-each` are like the arguments to `map`, but `for-each` calls *procedure* for its side effects rather than for its values. Unlike `map`, `for-each` is guaranteed to call *procedure* on the elements of the *list*s in order from the first element(s) to the last, and the value returned by `for-each` is unspecified. If more than one *list* is given and not all lists have the same length, `for-each` terminates when the shortest *list* runs out. The *list*s can be circular, but it is an error if all of them are circular.

It is an error for *procedure* to mutate any of the lists.

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                                ⇒ #(0 1 4 9 16)
```

`any` *predicate list list ...* [SRFI 1 procedure]

Applies *predicate* across the *list*s, returning true if *predicate* returns true on any application.

If there are n list arguments $list_1$ ... $list_n$, then *predicate* must be a procedure taking n arguments and returning a boolean result.

`any` applies *predicate* to the first elements of the *list* parameters. If this application returns a true value, `any` immediately returns that value. Otherwise, it iterates, applying *predicate* to the second elements of the *list* parameters, then the third, and so forth. The iteration stops when a true value is produced or one of the lists runs out of values; in the latter case, `any` returns `#f`. The application of *predicate* to the last element of the *list*s is a tail call.

Note the difference between `find` and `any`—`find` returns the element that satisfied the predicate; `any` returns the true value that the *predicate* produced.

Like `every`, `any`'s name does not end with a question mark—this is to indicate that it does not return a simple boolean (`#t` or `#f`), but a general value.

```
(any integer? '(a 3 b 2.7))   => #t
(any integer? '(a 3.1 b 2.7)) => #f
(any < '(3 1 4 1 5)
       '(2 7 1 8 2)) => #t
```

`every` *predicate list list ...* [SRFI 1 procedure]

Applies *predicate* across the *list*s, returning true if *predicate* returns true on every application.

If there are n list arguments $list_1$ ... $list_n$, then *predicate* must be a procedure taking n arguments and returning a boolean result.

`every` applies *predicate* to the first elements of the *list* parameters. If this application returns false, `every` immediately returns false. Otherwise, it iterates, applying *predicate* to the second elements of the *list* parameters, then the third, and so forth. The iteration stops when a false value is produced or one of the *list*s runs out of values. In the latter case, `every` returns the true value produced by its final application of *predicate*. The application of *predicate* to the last element of the *list*s is a tail call.

If one of the *list*s has no elements, `every` simply returns `#t`.

Like `any`, `every`'s name does not end with a question mark—this is to indicate that it does not return a simple boolean (`#t` or `#f`), but a general value.

7.8 Folding of Lists

`fold` *kons knil clist*$_1$ *clist*$_2$ ... [SRFI 1 procedure]

The fundamental list iterator.

First, consider the single list-parameter case. If *clist*$_1$ = `(`e_1 e_2 ... e_n`)`, then this procedure returns

```
(kons e_n ... (kons e_2 (kons e_1 knil)) ...)
```

That is, it obeys the (tail) recursion

```
(fold kons knil lis) = (fold kons (kons (car lis) knil) (cdr lis))
(fold kons knil '()) = knil
```

Examples:

```
(fold + 0 lis)                  ; Add up the elements of LIS.
```

```
(fold cons '() lis)             ; Reverse LIS.

(fold cons tail rev-head)       ; See APPEND-REVERSE.

;; How many symbols in LIS?
(fold (lambda (x count) (if (symbol? x) (+ count 1) count))
      0
      lis)

;; Length of the longest string in LIS:
(fold (lambda (s max-len) (max max-len (string-length s)))
      0
      lis)
```

If *n* list arguments are provided, then the *kons* procedure must take *n*+1 parameters: one element from each list, and the "seed" or fold state, which is initially *knil*. The fold operation terminates when the shortest list runs out of values:

```
(fold cons* '() '(a b c) '(1 2 3 4 5)) => (c 3 b 2 a 1)
```

At least one of the list arguments must be finite.

`fold-right` *kons knil clist*$_1$ *clist*$_2$ *...* [SRFI 1 procedure]

The fundamental list recursion operator.

First, consider the single list-parameter case. If *clist*$_1$ = (*e*$_1$ *e*$_2$ *... e*$_n$), then this procedure returns

(*kons e*$_1$ (*kons e*$_2$ *...* (*kons e*$_n$ *knil*)))

That is, it obeys the recursion

```
(fold-right kons knil lis) = (kons (car lis) (fold-right kons knil (cdr lis)))
(fold-right kons knil '()) = knil
```

Examples:

```
(fold-right cons '() lis)               ; Copy LIS.

;; Filter the even numbers out of LIS.
(fold-right (lambda (x l) (if (even? x) (cons x l) l)) '() lis))
```

If *n* list arguments are provided, then the *kons* function must take *n*+1 parameters: one element from each list, and the "seed" or fold state, which is initially *knil*. The fold operation terminates when the shortest list runs out of values:

```
(fold-right cons* '() '(a b c) '(1 2 3 4 5)) => (a 1 b 2 c 3)
```

At least one of the list arguments must be finite.

`fold-left` *proc knil list* [obsolete procedure]

Deprecated, use `fold` instead. Equivalent to

```
(fold (lambda (acc elt) (proc elt acc)) knil list)
```

`reduce` *f ridentity list* [SRFI 1 procedure]

`reduce` is a variant of `fold`.

ridentity should be a "right identity" of the procedure *f*—that is, for any value *x* acceptable to *f*,

```
(f x ridentity) = x
```

`reduce` has the following definition:

If *list* = `()`, return *ridentity*;
Otherwise, return `(fold f (car list) (cdr list))`.

...in other words, we compute `(fold f ridentity list)`.

Note that *ridentity* is used only in the empty-list case. You typically use `reduce` when applying *f* is expensive and you'd like to avoid the extra application incurred when `fold` applies *f* to the head of list and the identity value, redundantly producing the same value passed in to *f*. For example, if *f* involves searching a file directory or performing a database query, this can be significant. In general, however, `fold` is useful in many contexts where `reduce` is not (consider the examples given in the `fold` definition—only one of the five `fold`s uses a function with a right identity. The other four may not be performed with `reduce`).

```
;; Take the max of a list of non-negative integers.
(reduce max 0 nums) ; i.e., (apply max 0 nums)
```

`reduce-right` *kons knil list* [SRFI 1 procedure]

`reduce-right` is the `fold-right` variant of `reduce`. It obeys the following definition:

```
(reduce-right f ridentity '()) = ridentity
(reduce-right f ridentity '(e1)) = (f e1 ridentity) = e1
(reduce-right f ridentity '(e1 e2 ...)) =
  (f e1 (reduce f ridentity '(e2 ...)))
```

...in other words, we compute `(fold-right f ridentity list)`.

```
;; Append a bunch of lists together.
;; I.e., (apply append list-of-lists)
(reduce-right append '() list-of-lists)
```

`reduce-left` *f ridentity list* [obsolete procedure]

Deprecated, use `reduce` instead. Equivalent to

```
(reduce (lambda (acc elt) (f elt acc)) ridentity list)
```

7.9 Miscellaneous List Operations

`circular-list` *object ...* [SRFI 1 procedure]
`make-circular-list` *k* [*element*] [procedure]

`circular-list` returns a circular list containing the given *object*s. `make-circular-list` returns a circular list of length *k*; if *element* is given, the returned list is filled with it, otherwise the elements are unspecified.

This procedure is like `list` except that the returned list is circular. `circular-list` could have been defined like this:

```
(define (circular-list . objects)
  (append! objects objects))
```

`circular-list` is compatible with SRFI 1, but extended so that it can be called with no arguments.

`reverse` *list* [standard procedure]

Returns a newly allocated list consisting of the top-level elements of *list* in reverse order.

```
(reverse '(a b c))                  ⇒ (c b a)
(reverse '(a (b c) d (e (f))))      ⇒ ((e (f)) d (b c) a)
```

`reverse!` *list* [SRFI 1 procedure]

Returns a list consisting of the top-level elements of *list* in reverse order. `reverse!` is like `reverse`, except that it destructively modifies *list*. Because the result may not be `eqv?` to *list*, it is desirable to do something like `(set! x (reverse! x))`.

`sort` *sequence procedure* [*key*] [procedure]
`merge-sort` *sequence procedure* [*key*] [procedure]
`quick-sort` *sequence procedure* [*key*] [procedure]

Sequence must be either a list or a vector. *Key*, if specified, must be a procedure of one argument that maps an element of *sequence* to a key fit for comparison by *procedure*; by default, *key* is the identity. *Procedure* must be a procedure of two arguments that defines a *total ordering* on the keys of *sequence*. In other words, if *x* and *y* are two distinct elements of *sequence*, then it must be the case that

```
(and (procedure (key x) (key y))
     (procedure (key y) (key x)))
     ⇒ #f
```

If *sequence* is a list (vector), `sort` returns a newly allocated list (vector) whose elements are those of *sequence*, except that they are rearranged to be sorted in the order defined by *procedure* and *key*. So, for example, if the elements of *sequence* are numbers, and *procedure* is `<`, then the resulting elements are sorted in monotonically nondecreasing order. Likewise, if *procedure* is `>`, the resulting elements are sorted in monotonically nonincreasing order. To be precise, if *x* and *y* are any two adjacent elements in the result, where *x* precedes *y*, it is the case that

```
(procedure (key y) (key x))
     ⇒ #f
```

Two sorting algorithms are implemented: `merge-sort` and `quick-sort`. The procedure `sort` is an alias for `merge-sort`. `Merge-sort` is stable, meaning that it preserves the order in *sequence* of elements which are equivalent under *procedure* and *key*; `quick-sort` is not stable, so it does not guarantee this.

See also the definition of `sort!`.

```
(merge-sort '((2 . foo) (2 . bar) (1 . baz) (3 . quux)) < car)
⇒ ((1 . baz) (2 . foo) (2 . bar) (3 . quux))

(quick-sort '((2 . foo) (2 . bar) (1 . baz) (3 . quux)) < car)
⇒ ((1 . baz) (2 . bar) (2 . foo) (3 . quux))
```

8 Vectors

Vectors are heterogenous structures whose elements are indexed by exact non-negative integers. A vector typically occupies less space than a list of the same length, and the average time required to access a randomly chosen element is typically less for the vector than for the list.

The *length* of a vector is the number of elements that it contains. This number is an exact non-negative integer that is fixed when the vector is created. The *valid indexes* of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Vectors are written using the notation `#(`*`object ...`*`)`. For example, a vector of length 3 containing the number zero in element 0, the list `(2 2 2 2)` in element 1, and the string `"Anna"` in element 2 can be written as

```
#(0 (2 2 2 2) "Anna")
```

Note that this is the external representation of a vector, not an expression evaluating to a vector. Like list constants, vector constants must be quoted:

```
'#(0 (2 2 2 2) "Anna")          ⇒  #(0 (2 2 2 2) "Anna")
```

A number of the vector procedures operate on subvectors. A *subvector* is a segment of a vector that is specified by two exact non-negative integers, *start* and *end*. *Start* is the index of the first element that is included in the subvector, and *end* is one greater than the index of the last element that is included in the subvector. Thus if *start* and *end* are the same, they refer to a null subvector, and if *start* is zero and *end* is the length of the vector, they refer to the entire vector. The *valid indexes* of a subvector are the exact integers between *start* inclusive and *end* exclusive.

8.1 Construction of Vectors

`make-vector` *k* [*object*] [procedure]

Returns a newly allocated vector of *k* elements. If *object* is specified, `make-vector` initializes each element of the vector to *object*. Otherwise the initial elements of the result are unspecified.

`vector` *object ...* [procedure]

Returns a newly allocated vector whose elements are the given arguments. `vector` is analogous to `list`.

```
(vector 'a 'b 'c)                    ⇒  #(a b c)
```

`vector-copy` *vector* [procedure]

Returns a newly allocated vector that is a copy of *vector*.

`list->vector` *list* [procedure]

Returns a newly allocated vector initialized to the elements of *list*. The inverse of `list->vector` is `vector->list`.

```
(list->vector '(dididit dah))        ⇒  #(dididit dah)
```

`string->vector` *string* [*start* [*end*]] [standard procedure]
`vector->string` *vector* [*start* [*end*]] [standard procedure]

It is an error if any element of *vector* is not a character.

The `vector->string` procedure returns a newly allocated string of the objects contained in the elements of *vector* between *start* and *end*. The `string->vector` procedure returns a newly created vector initialized to the elements of the string *string* between *start* and *end*.

In both procedures, order is preserved.

```
(string->vector "ABC")                  ⇒  #(#\A #\B #\C)
(vector->string #(#\1 #\2 #\3)          ⇒  "123"
```

`make-initialized-vector` *k initialization* [procedure]

Similar to `make-vector`, except that the elements of the result are determined by calling the procedure *initialization* on the indices. For example:

```
(make-initialized-vector 5 (lambda (x) (* x x)))
     ⇒  #(0 1 4 9 16)
```

`vector-grow` *vector k* [procedure]

K must be greater than or equal to the length of *vector*. Returns a newly allocated vector of length *k*. The first `(vector-length vector)` elements of the result are initialized from the corresponding elements of *vector*. The remaining elements of the result are unspecified.

`vector-map` *procedure vector vector* ... [standard procedure]

It is an error if *procedure* does not accept as many arguments as there are *vector*s and return a single value.

The `vector-map` procedure applies *procedure* element-wise to the elements of the *vector*s and returns a vector of the results, in order. If more than one *vector* is given and not all vectors are the same length, `vector-map` terminates when the shortest vector runs out. The dynamic order in which *procedure* is applied to the elements of the *vector*s is unspecified. If multiple returns occur from `vector-map`, the values returned by earlier returns are not mutated.

```
(vector-map cadr '#((a b) (d e) (g h))) ⇒  #(b e h)
(vector-map (lambda (n) (expt n n)) '#(1 2 3 4 5))
                                        ⇒  #(1 4 27 256 3125)
(vector-map + '#(1 2 3) '#(4 5 6 7))    ⇒  #(5 7 9)
(let ((count 0))
  (vector-map (lambda (ignored)
                (set! count (+ count 1))
                count)
              '#(a b)))                 ⇒ #(1 2) or #(2 1)
```

`vector-for-each` *procedure vector vector* ... [standard procedure]

It is an error if *procedure* does not accept as many arguments as there are *vector*s.

The arguments to `vector-for-each` are like the arguments to `vector-map`, but `vector-for-each` calls *procedure* for its side effects rather than for its values. Unlike `vector-map`, `vector-for-each` is guaranteed to call *procedure* on the elements of

the *vector*s in order from the first element(s) to the last, and the value returned by `vector-for-each` is unspecified. If more than one *vector* is given and not all vectors have the same length, `vector-for-each` terminates when the shortest *vector* runs out. It is an error for *procedure* to mutate any of the vectors.

8.2 Selecting Vector Components

`vector?` *object* [procedure]
Returns `#t` if *object* is a vector; otherwise returns `#f`.

`vector-length` *vector* [procedure]
Returns the number of elements in *vector*.

`vector-ref` *vector k* [procedure]
Returns the contents of element *k* of *vector*. *K* must be a valid index of *vector*.

```
(vector-ref '#(1 1 2 3 5 8 13 21) 5)    ⇒  8
```

`vector-set!` *vector k object* [procedure]
Stores *object* in element *k* of *vector* and returns an unspecified value. *K* must be a valid index of *vector*.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
     ⇒  #(0 ("Sue" "Sue") "Anna")
```

`vector-first` *vector* [procedure]
`vector-second` *vector* [procedure]
`vector-third` *vector* [procedure]
`vector-fourth` *vector* [procedure]
`vector-fifth` *vector* [procedure]
`vector-sixth` *vector* [procedure]
`vector-seventh` *vector* [procedure]
`vector-eighth` *vector* [procedure]
These procedures access the first several elements of *vector* in the obvious way. It is an error if the implicit index of one of these procedurs is not a valid index of *vector*.

`vector-binary-search` *vector key<? unwrap-key key* [procedure]
Searches *vector* for an element with a key matching *key*, returning the element if one is found or `#f` if none. The search operation takes time proportional to the logarithm of the length of *vector*. *Unwrap-key* must be a procedure that maps each element of *vector* to a key. *Key<?* must be a procedure that implements a total ordering on the keys of the elements.

```
(define (translate number)
  (vector-binary-search '#((1 . i)
                           (2 . ii)
                           (3 . iii)
                           (6 . vi))
                        < car number))
(translate 2)  ⇒  (2 . ii)
(translate 4)  ⇒  #F
```

8.3 Cutting Vectors

`subvector` *vector start end* [procedure]
Returns a newly allocated vector that contains the elements of *vector* between index *start* (inclusive) and *end* (exclusive).

`vector-head` *vector end* [procedure]
Equivalent to

```
(subvector vector 0 end)
```

`vector-tail` *vector start* [procedure]
Equivalent to

```
(subvector vector start (vector-length vector))
```

8.4 Modifying Vectors

`vector-fill!` *vector object* [procedure]
`subvector-fill!` *vector start end object* [procedure]
Stores *object* in every element of the vector (subvector) and returns an unspecified value.

`subvector-move-left!` *vector1 start1 end1 vector2 start2* [procedure]
`subvector-move-right!` *vector1 start1 end1 vector2 start2* [procedure]
Destructively copies the elements of *vector1*, starting with index *start1* (inclusive) and ending with *end1* (exclusive), into *vector2* starting at index *start2* (inclusive). *Vector1*, *start1*, and *end1* must specify a valid subvector, and *start2* must be a valid index for *vector2*. The length of the source subvector must not exceed the length of *vector2* minus the index *start2*.

The elements are copied as follows (note that this is only important when *vector1* and *vector2* are `eqv?`):

`subvector-move-left!`
The copy starts at the left end and moves toward the right (from smaller indices to larger). Thus if *vector1* and *vector2* are the same, this procedure moves the elements toward the left inside the vector.

`subvector-move-right!`
The copy starts at the right end and moves toward the left (from larger indices to smaller). Thus if *vector1* and *vector2* are the same, this procedure moves the elements toward the right inside the vector.

`sort!` *vector procedure key* [procedure]
`merge-sort!` *vector procedure key* [procedure]
`quick-sort!` *vector procedure key* [procedure]

Key, if specified, must be a procedure of one argument that maps an element of *sequence* to a key fit for comparison by *procedure*; by default, *key* is the identity. *Procedure* must be a procedure of two arguments that defines a *total ordering* on the keys of *vector*. The elements of *vector* are rearranged so that they are sorted in the order defined by *procedure* and *key*. The elements are rearranged in place, that is, *vector* is destructively modified so that its elements are in the new order.

`sort!` returns *vector* as its value.

Two sorting algorithms are implemented: `merge-sort!` and `quick-sort!`. The procedure `sort!` is an alias for `merge-sort!`.

See also the definition of `sort`.

9 Bit Strings

A *bit string* is a sequence of bits. Bit strings can be used to represent sets or to manipulate binary data. The elements of a bit string are numbered from zero up to the number of bits in the string less one, in *right to left order*, (the rightmost bit is numbered zero). When you convert from a bit string to an integer, the zero-th bit is associated with the zero-th power of two, the first bit is associated with the first power, and so on.

Bit strings are encoded very densely in memory. Each bit occupies exactly one bit of storage, and the overhead for the entire bit string is bounded by a small constant. However, accessing a bit in a bit string is slow compared to accessing an element of a vector or character string. If performance is of overriding concern, it is better to use character strings to store sets of boolean values even though they occupy more space.

The *length* of a bit string is the number of bits that it contains. This number is an exact non-negative integer that is fixed when the bit string is created. The *valid indexes* of a bit string are the exact non-negative integers less than the length of the bit string.

Bit strings may contain zero or more bits. They are not limited by the length of a machine word. In the printed representation of a bit string, the contents of the bit string are preceded by '`#*`'. The contents are printed starting with the most significant bit (highest index).

Note that the external representation of bit strings uses a bit ordering that is the reverse of the representation for bit strings in Common Lisp. It is likely that MIT/GNU Scheme's representation will be changed in the future, to be compatible with Common Lisp. For the time being this representation should be considered a convenience for viewing bit strings rather than a means of entering them as data.

```
#*11111
#*1010
#*00000000
#*
```

All of the bit-string procedures are MIT/GNU Scheme extensions.

9.1 Construction of Bit Strings

`make-bit-string` *k initialization* [procedure]
Returns a newly allocated bit string of length *k*. If *initialization* is `#f`, the bit string is filled with 0 bits; otherwise, the bit string is filled with 1 bits.

```
(make-bit-string 7 #f)                    ⇒  #*0000000
```

`bit-string-allocate` *k* [procedure]
Returns a newly allocated bit string of length *k*, but does not initialize it.

`bit-string-copy` *bit-string* [procedure]
Returns a newly allocated copy of *bit-string*.

9.2 Selecting Bit String Components

`bit-string?` *object* [procedure]
Returns `#t` if *object* is a bit string; otherwise returns `#f`.

`bit-string-length` *bit-string* [procedure]
Returns the length of *bit-string*.

`bit-string-ref` *bit-string k* [procedure]
Returns `#t` if the *k*th bit is 1; otherwise returns `#f`. *K* must be a valid index of *bit-string*.

`bit-string-set!` *bit-string k* [procedure]
Sets the *k*th bit in *bit-string* to 1 and returns an unspecified value. *K* must be a valid index of *bit-string*.

`bit-string-clear!` *bit-string k* [procedure]
Sets the *k*th bit in *bit-string* to 0 and returns an unspecified value. *K* must be a valid index of *bit-string*.

`bit-substring-find-next-set-bit` *bit-string start end* [procedure]
Returns the index of the first occurrence of a set bit in the substring of *bit-string* from *start* (inclusive) to *end* (exclusive). If none of the bits in the substring are set `#f` is returned. The index returned is relative to the whole bit string, not substring.

The following procedure uses `bit-substring-find-next-set-bit` to find all the set bits and display their indexes:

```
(define (scan-bitstring bs)
  (let ((end (bit-string-length bs)))
    (let loop ((start 0))
      (let ((next
             (bit-substring-find-next-set-bit bs start end)))
        (if next
            (begin
              (write-line next)
              (if (< next end)
                  (loop (+ next 1)))))))))
```

9.3 Cutting and Pasting Bit Strings

`bit-string-append` *bit-string-1 bit-string-2* [procedure]
Appends the two bit string arguments, returning a newly allocated bit string as its result. In the result, the bits copied from *bit-string-1* are less significant (smaller indices) than those copied from *bit-string-2*.

`bit-substring` *bit-string start end* [procedure]
Returns a newly allocated bit string whose bits are copied from *bit-string*, starting at index *start* (inclusive) and ending at *end* (exclusive).

9.4 Bitwise Operations on Bit Strings

`bit-string-zero?` *bit-string* [procedure]
Returns `#t` if *bit-string* contains only 0 bits; otherwise returns `#f`.

`bit-string=?` *bit-string-1 bit-string-2* [procedure]
Compares the two bit string arguments and returns `#t` if they are the same length and contain the same bits; otherwise returns `#f`.

`bit-string-not` *bit-string* [procedure]
Returns a newly allocated bit string that is the bitwise-logical negation of *bit-string*.

`bit-string-movec!` *target-bit-string bit-string* [procedure]
The destructive version of `bit-string-not`. The arguments *target-bit-string* and *bit-string* must be bit strings of the same length. The bitwise-logical negation of *bit-string* is computed and the result placed in *target-bit-string*. The value of this procedure is unspecified.

`bit-string-and` *bit-string-1 bit-string-2* [procedure]
Returns a newly allocated bit string that is the bitwise-logical "and" of the arguments. The arguments must be bit strings of identical length.

`bit-string-andc` *bit-string-1 bit-string-2* [procedure]
Returns a newly allocated bit string that is the bitwise-logical "and" of *bit-string-1* with the bitwise-logical negation of *bit-string-2*. The arguments must be bit strings of identical length.

`bit-string-or` *bit-string-1 bit-string-2* [procedure]
Returns a newly allocated bit string that is the bitwise-logical "inclusive or" of the arguments. The arguments must be bit strings of identical length.

`bit-string-xor` *bit-string-1 bit-string-2* [procedure]
Returns a newly allocated bit string that is the bitwise-logical "exclusive or" of the arguments. The arguments must be bit strings of identical length.

`bit-string-and!` *target-bit-string bit-string* [procedure]
`bit-string-or!` *target-bit-string bit-string* [procedure]
`bit-string-xor!` *target-bit-string bit-string* [procedure]
`bit-string-andc!` *target-bit-string bit-string* [procedure]
These are destructive versions of the above operations. The arguments *target-bit-string* and *bit-string* must be bit strings of the same length. Each of these procedures performs the corresponding bitwise-logical operation on its arguments, places the result into *target-bit-string*, and returns an unspecified result.

9.5 Modification of Bit Strings

`bit-string-fill!` *bit-string initialization* [procedure]
Fills *bit-string* with zeroes if *initialization* is `#f`; otherwise fills *bit-string* with ones. Returns an unspecified value.

`bit-string-move!` *target-bit-string bit-string* [procedure]
Moves the contents of *bit-string* into *target-bit-string*. Both arguments must be bit strings of the same length. The results of the operation are undefined if the arguments are the same bit string.

`bit-substring-move-right!` *bit-string-1 start1 end1 bit-string-2 start2* [procedure]
Destructively copies the bits of *bit-string-1*, starting at index *start1* (inclusive) and ending at *end1* (exclusive), into *bit-string-2* starting at index *start2* (inclusive). *Start1* and *end1* must be valid substring indices for *bit-string-1*, and *start2* must be a valid index for *bit-string-2*. The length of the source substring must not exceed the length of *bit-string-2* minus the index *start2*.

The bits are copied starting from the MSB and working towards the LSB; the direction of copying only matters when *bit-string-1* and *bit-string-2* are `eqv?`.

9.6 Integer Conversions of Bit Strings

`unsigned-integer->bit-string` *length integer* [procedure]
Both *length* and *integer* must be exact non-negative integers. Converts *integer* into a newly allocated bit string of *length* bits. Signals an error of type `condition-type:bad-range-argument` if *integer* is too large to be represented in *length* bits.

`signed-integer->bit-string` *length integer* [procedure]
Length must be an exact non-negative integer, and *integer* may be any exact integer. Converts *integer* into a newly allocated bit string of *length* bits, using two's complement encoding for negative numbers. Signals an error of type `condition-type:bad-range-argument` if *integer* is too large to be represented in *length* bits.

`bit-string->unsigned-integer` *bit-string* [procedure]
`bit-string->signed-integer` *bit-string* [procedure]
Converts *bit-string* into an exact integer. `bit-string->signed-integer` regards *bit-string* as a two's complement representation of a signed integer, and produces an integer of like sign and absolute value. `bit-string->unsigned-integer` regards *bit-string* as an unsigned quantity and converts to an integer accordingly.

10 Miscellaneous Datatypes

10.1 Booleans

The *boolean objects* are *true* and *false*. The boolean constant true is written as '`#t`', and the boolean constant false is written as '`#f`'.

The primary use for boolean objects is in the conditional expressions `if`, `cond`, `and`, and `or`; the behavior of these expressions is determined by whether objects are true or false. These expressions count only `#f` as false. They count everything else, including `#t`, pairs, symbols, numbers, strings, vectors, and procedures as true (but see Section 1.2.5 [True and False], page 8).

Programmers accustomed to other dialects of Lisp should note that Scheme distinguishes `#f` and the empty list from the symbol `nil`. Similarly, `#t` is distinguished from the symbol `t`. In fact, the boolean objects (and the empty list) are not symbols at all.

Boolean constants evaluate to themselves, so you don't need to quote them.

```
#t                                      ⇒  #t
#f                                      ⇒  #f
'#f                                     ⇒  #f
t                                       error  Unbound variable
```

`false` [variable]
`true` [variable]

These variables are bound to the objects `#f` and `#t` respectively. The compiler, given the `usual-integrations` declaration, replaces references to these variables with their respective values.

Note that the symbol `true` is not equivalent to `#t`, and the symbol `false` is not equivalent to `#f`.

`boolean?` *object* [standard procedure]

Returns `#t` if *object* is either `#t` or `#f`; otherwise returns `#f`.

```
(boolean? #f)                           ⇒  #t
(boolean? 0)                            ⇒  #f
```

`not` *object* [standard procedure]
`false?` *object* [procedure]

These procedures return `#t` if *object* is false; otherwise they return `#f`. In other words they *invert* boolean values. These two procedures have identical semantics; their names are different to give different connotations to the test.

```
(not #t)                                ⇒  #f
(not 3)                                 ⇒  #f
(not (list 3))                          ⇒  #f
(not #f)                                ⇒  #t
```

`procedure` *boolean=? boolean1 boolean2 boolean3* [extended standard procedure]
...

This predicate is true iff the *boolean* args are either all true or all false.

Implementation note: The standard requires this procedure's arguments to satisfy `boolean?`, but MIT/GNU Scheme allows any object to be an argument.

`boolean/and` *object ...* [procedure]
This procedure returns `#t` if none of its arguments are `#f`. Otherwise it returns `#f`.

`boolean/or` *object ...* [procedure]
This procedure returns `#f` if all of its arguments are `#f`. Otherwise it returns `#t`.

10.2 Symbols

MIT/GNU Scheme provides two types of symbols: *interned* and *uninterned*. Interned symbols are far more common than uninterned symbols, and there are more ways to create them. Interned symbols have an external representation that is recognized by the procedure `read`; uninterned symbols do not.[1]

Interned symbols have an extremely useful property: any two interned symbols whose names are the same, in the sense of `string=?`, are the same object (i.e. they are `eq?` to one another). The term *interned* refers to the process of *interning* by which this is accomplished. Uninterned symbols do not share this property.

The names of interned symbols are not distinguished by their alphabetic case. Because of this, MIT/GNU Scheme converts all alphabetic characters in the name of an interned symbol to a specific case (lower case) when the symbol is created. When the name of an interned symbol is referenced (using `symbol->string`) or written (using `write`) it appears in this case. It is a bad idea to depend on the name being lower case. In fact, it is preferable to take this one step further: don't depend on the name of a symbol being in a uniform case.

The rules for writing an interned symbol are the same as the rules for writing an identifier (see Section 1.3.3 [Identifiers], page 10). Any interned symbol that has been returned as part of a literal expression, or read using the `read` procedure and subsequently written out using the `write` procedure, will read back in as the identical symbol (in the sense of `eq?`).

Usually it is also true that reading in an interned symbol that was previously written out produces the same symbol. An exception are symbols created by the procedures `string->symbol` and `intern`; they can create symbols for which this write/read invariance may not hold because the symbols' names contain special characters or letters in the non-standard case.[2]

---

[1] In older dialects of Lisp, uninterned symbols were fairly important. This was true because symbols were complicated data structures: in addition to having value cells (and sometimes, function cells), these structures contained *property lists*. Because of this, uninterned symbols were often used merely for their property lists — sometimes an uninterned symbol used this way was referred to as a *disembodied property list*. In MIT/GNU Scheme, symbols do not have property lists, or any other components besides their names. There is a different data structure similar to disembodied property lists: one-dimensional tables (see Section 11.2 [1D Tables], page 183). For these reasons, uninterned symbols are not very useful in MIT/GNU Scheme. In fact, their primary purpose is to simplify the generation of unique variable names in programs that generate Scheme code.

[2] MIT/GNU Scheme reserves a specific set of interned symbols for its own use. If you use these reserved symbols it is possible that you could break specific pieces of software that depend on them. The reserved symbols all have names beginning with the characters '`#[`' and ending with the character '`]`'; thus none of these symbols can be read by the procedure `read` and hence are not likely to be used by accident. For example, `(intern "#[unnamed-procedure]")` produces a reserved symbol.

The external representation for uninterned symbols is special, to distinguish them from interned symbols and prevent them from being recognized by the `read` procedure:

```
(string->uninterned-symbol "foo")
     ⇒  #[uninterned-symbol 30 foo]
```

In this section, the procedures that return symbols as values will either always return interned symbols, or always return uninterned symbols. The procedures that accept symbols as arguments will always accept either interned or uninterned symbols, and do not distinguish the two.

`symbol?` *object* [procedure]

Returns `#t` if *object* is a symbol, otherwise returns `#f`.

```
(symbol? 'foo)                          ⇒  #t
(symbol? (car '(a b)))                  ⇒  #t
(symbol? "bar")                         ⇒  #f
```

`symbol->string` *symbol* [procedure]

Returns the name of *symbol* as a string. If *symbol* was returned by `string->symbol`, the value of this procedure will be identical (in the sense of `string=?`) to the string that was passed to `string->symbol`. It is an error to apply mutation procedures such as `string-set!` to strings returned by this procedure.

```
(symbol->string 'flying-fish)           ⇒  "flying-fish"
(symbol->string 'Martin)                ⇒  "martin"
(symbol->string (string->symbol "Malvina"))
                                        ⇒  "Malvina"
```

Note that two distinct uninterned symbols can have the same name.

`intern` *string* [procedure]

Returns the interned symbol whose name is *string*. Converts *string* to the standard alphabetic case before generating the symbol. This is the preferred way to create interned symbols, as it guarantees the following independent of which case the implementation uses for symbols' names:

```
(eq? 'bitBlt (intern "bitBlt")) ⇒     #t
```

The user should take care that *string* obeys the rules for identifiers (see Section 1.3.3 [Identifiers], page 10), otherwise the resulting symbol cannot be read as itself.

`intern-soft` *string* [procedure]

Returns the interned symbol whose name is *string*. Converts *string* to the standard alphabetic case before generating the symbol. If no such interned symbol exists, returns `#f`.

This is exactly like `intern`, except that it will not create an interned symbol, but only returns symbols that already exist.

`string->symbol` *string* [procedure]

Returns the interned symbol whose name is *string*. Although you can use this procedure to create symbols with names containing special characters or lowercase letters,

it's usually a bad idea to create such symbols because they cannot be read as themselves. See `symbol->string`.

```
(eq? 'mISSISSIppi 'mississippi)         ⇒  #t
(string->symbol "mISSISSIppi")
     ⇒  the symbol with the name "mISSISSIppi"
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒  #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))     ⇒  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol
              "K. Harper, M.D.")))      ⇒  #t
```

`string->uninterned-symbol` *string* [procedure]

Returns a newly allocated uninterned symbol whose name is *string*. It is unimportant what case or characters are used in *string*.

Note: this is the fastest way to make a symbol.

`generate-uninterned-symbol` [*object*] [procedure]

Returns a newly allocated uninterned symbol that is guaranteed to be different from any other object. The symbol's name consists of a prefix string followed by the (exact non-negative integer) value of an internal counter. The counter is initially zero, and is incremented after each call to this procedure.

The optional argument *object* is used to control how the symbol is generated. It may take one of the following values:

- If *object* is omitted or `#f`, the prefix is `"G"`.
- If *object* is an exact non-negative integer, the internal counter is set to that integer prior to generating the result.
- If *object* is a string, it is used as the prefix.
- If *object* is a symbol, its name is used as the prefix.

```
(generate-uninterned-symbol)
     ⇒  #[uninterned-symbol 31 G0]
(generate-uninterned-symbol)
     ⇒  #[uninterned-symbol 32 G1]
(generate-uninterned-symbol 'this)
     ⇒  #[uninterned-symbol 33 this2]
(generate-uninterned-symbol)
     ⇒  #[uninterned-symbol 34 G3]
(generate-uninterned-symbol 100)
     ⇒  #[uninterned-symbol 35 G100]
(generate-uninterned-symbol)
     ⇒  #[uninterned-symbol 36 G101]
```

`symbol-append` *symbol* . . . [procedure]

Returns the interned symbol whose name is formed by concatenating the names of the given symbols. This procedure preserves the case of the names of its arguments,

so if one or more of the arguments' names has non-standard case, the result will also have non-standard case.

```
(symbol-append 'foo- 'bar)                ⇒  foo-bar
;; the arguments may be uninterned:
(symbol-append 'foo- (string->uninterned-symbol "baz"))
                                          ⇒  foo-baz
;; the result has the same case as the arguments:
(symbol-append 'foo- (string->symbol "BAZ"))     ⇒  foo-BAZ
```

`symbol-hash` *symbol* [procedure]

Returns a hash number for *symbol*, which is computed by calling `string-hash` on *symbol*'s name. The hash number is an exact non-negative integer.

`symbol-hash-mod` *symbol modulus* [procedure]

Modulus must be an exact positive integer. Equivalent to

```
(modulo (symbol-hash symbol) modulus)
```

This procedure is provided for convenience in constructing hash tables. However, it is normally preferable to use `make-strong-eq-hash-table` to build hash tables keyed by symbols, because `eq?` hash tables are much faster.

`symbol<?` *symbol1 symbol2* [procedure]

This procedure computes a total order on symbols. It is equivalent to

```
(string<? (symbol->string symbol1)
          (symbol->string symbol2))
```

10.3 Parameters

Parameters are objects that can be bound to new values for the duration of a dynamic extent. See Section 2.3 [Dynamic Binding], page 20.

`make-parameter` *init* [*converter*] [procedure]
`make-unsettable-parameter` *init* [*converter*] [procedure]

Returns a newly allocated parameter object, which is a procedure that accepts zero arguments and returns the value associated with the parameter object. Initially this value is the value of `(converter init)`, or of *init* if the conversion procedure *converter* is not specified. The associated value can be temporarily changed using the `parameterize` special form (see [parameterize], page 20).

The `make-parameter` procedure is standardized by SRFI 39 and by R7RS, while `make-unsettable-parameter` is an MIT/GNU Scheme extension.

`make-settable-parameter` *init* [*converter*] [procedure]

This procedure is like `make-parameter`, except that the returned parameter object may also be assigned by passing it an argument. Note that an assignment to a settable parameter affects only the extent of its current binding.

`make-settable-parameter` is an MIT/GNU Scheme extension.

`parameterize*` *bindings thunk* [procedure]
: *Bindings* should be an alist associating parameter objects with new values. Returns the value of *thunk* while the parameters are dynamically bound to the values.

  Note that the `parameterize` special form expands into a call to this procedure. `parameterize*` is an MIT/GNU Scheme extension.

10.3.1 Cells

A *cell* object is very similar to a parameter but is not implemented in multi-processing worlds and thus is **deprecated**. Parameters should be used instead.

`cell?` *object* [procedure]
: Returns `#t` if *object* is a cell; otherwise returns `#f`.

`make-cell` *object* [procedure]
: Returns a newly allocated cell whose contents is *object*.

`cell-contents` *cell* [procedure]
: Returns the current contents of *cell*.

`set-cell-contents!` *cell object* [procedure]
: Alters the contents of *cell* to be *object*. Returns an unspecified value.

`bind-cell-contents!` *cell object thunk* [procedure]
: Alters the contents of *cell* to be *object*, calls *thunk* with no arguments, then restores the original contents of *cell* and returns the value returned by *thunk*. This is completely equivalent to dynamic binding of a variable, including the behavior when continuations are used (see Section 2.3 [Dynamic Binding], page 20).

10.4 Records

MIT/GNU Scheme provides a *record* abstraction, which is a simple and flexible mechanism for building structures with named components. Records can be defined and accessed using the procedures defined in this section. A less flexible but more concise way to manipulate records is to use the `define-structure` special form (see Section 2.10 [Structure Definitions], page 31).

`make-record-type` *type-name field-names* [procedure]
: Returns a *record-type descriptor*, a value representing a new data type, disjoint from all others. The *type-name* argument must be a string, but is only used for debugging purposes (such as the printed representation of a record of the new type). The *field-names* argument is a list of symbols naming the *fields* of a record of the new type. It is an error if the list contains any duplicates. It is unspecified how record-type descriptors are represented.

`record-constructor` *record-type* [*field-names*] [procedure]
: Returns a procedure for constructing new members of the type represented by *record-type*. 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 the list of *field-names* 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 *record-type*; 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-keyword-constructor` *record-type* [procedure]
Returns a procedure for constructing new members of the type represented by *record-type*. The returned procedure accepts arguments in a *keyword list*, which is an alternating sequence of names and values. In other words, the number of arguments must be a multiple of two, and every other argument, starting with the first argument, must be a symbol that is one of the field names for *record-type*.

The returned procedure may be called with a keyword list that contains multiple instances of the same keyword. In this case, the leftmost instance is used and the other instances are ignored. This allows keyword lists to be accumulated using `cons` or `cons*`, and new bindings added to the front of the list override old bindings at the end.

`record-predicate` *record-type* [procedure]
Returns a procedure for testing membership in the type represented by *record-type*. The returned procedure accepts exactly one argument and returns `#t` if the argument is a member of the indicated record type; it returns `#f` otherwise.

`record-accessor` *record-type field-name* [procedure]
Returns a procedure for reading the value of a particular field of a member of the type represented by *record-type*. 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 *record-type*.

`record-modifier` *record-type field-name* [procedure]
Returns a procedure for writing the value of a particular field of a member of the type represented by *record-type*. 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 *record-type*.

`record?` *object* [procedure]
Returns `#t` if *object* is a record of any type and `#f` otherwise. Note that `record?` may be true of any Scheme value; of course, if it returns `#t` for some particular value, then `record-type-descriptor` is applicable to that value and returns an appropriate descriptor.

`record-type-descriptor` *record* [procedure]
Returns the record-type descriptor representing the type of *record*. That is, for example, if the returned descriptor were passed to `record-predicate`, the resulting predicate would return `#t` when passed *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 *record*.

`record-type?` *object* [procedure]
Returns `#t` if *object* is a record-type descriptor; otherwise returns `#f`.

`record-type-name` *record-type* [procedure]
Returns the type name associated with the type represented by *record-type*. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *record-type*.

`record-type-field-names` *record-type* [procedure]
Returns a list of the symbols naming the fields in members of the type represented by *record-type*. The returned value is `equal?` to the *field-names* argument given in the call to `make-record-type` that created the type represented by *record-type*.[3]

10.5 Promises

`delay` *expression* [special form]
The `delay` construct is used together with the procedure `force` to implement *lazy evaluation* or *call by need*. `(delay` *`expression`*`)` returns an object called a *promise* which at some point in the future may be asked (by the `force` procedure) to evaluate *expression* and deliver the resulting value.

`force` *promise* [procedure]
Forces the value of *promise*. If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached (or "memoized") so that if it is forced a second time, the previously computed value is returned without any recomputation.

```
(force (delay (+ 1 2)))                 ⇒  3

(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))           ⇒  (3 3)

(define head car)

(define tail
  (lambda (stream)
    (force (cdr stream))))

(define a-stream
  (letrec ((next
            (lambda (n)
              (cons n (delay (next (+ n 1)))))))
    (next 0)))

(head (tail (tail a-stream)))           ⇒  2
```

[3] In MIT/GNU Scheme, the returned list is always newly allocated.

`promise?` *object* [procedure]
Returns `#t` if *object* is a promise; otherwise returns `#f`.

`promise-forced?` *promise* [procedure]
Returns `#t` if *promise* has been forced and its value cached; otherwise returns `#f`.

`promise-value` *promise* [procedure]
If *promise* has been forced and its value cached, this procedure returns the cached value. Otherwise, an error is signalled.

`force` and `delay` are mainly intended for programs written in functional style. The following examples should not be considered to illustrate good programming style, but they illustrate the property that the value of a promise is computed at most once.

```
(define count 0)

(define p
  (delay
   (begin
     (set! count (+ count 1))
     (* x 3))))

(define x 5)

count                                   ⇒  0
p                                       ⇒  #[promise 54]
(force p)                               ⇒  15
p                                       ⇒  #[promise 54]
count                                   ⇒  1
(force p)                               ⇒  15
count                                   ⇒  1
```

Here is a possible implementation of `delay` and `force`. We define the expression

```
(delay expression)
```

to have the same meaning as the procedure call

```
(make-promise (lambda () expression))
```

where `make-promise` is defined as follows:

```
(define make-promise
  (lambda (proc)
    (let ((already-run? #f)
          (result #f))
      (lambda ()
        (cond ((not already-run?)
               (set! result (proc))
               (set! already-run? #t)))
        result))))
```

Promises are implemented here as procedures of no arguments, and `force` simply calls its argument.

```
(define force
  (lambda (promise)
    (promise)))
```

Various extensions to this semantics of `delay` and `force` are supported in some implementations (none of these are currently supported in MIT/GNU Scheme):

- Calling `force` on an object that is not a promise may simply return the object.
- It may be the case that there is no means by which a promise can be operationally distinguished from its forced value. That is, expressions like the following may evaluate to either `#t` or `#f`, depending on the implementation:

```
(eqv? (delay 1) 1)              ⇒  unspecified
(pair? (delay (cons 1 2)))      ⇒  unspecified
```

- Some implementations will implement "implicit forcing", where the value of a promise is forced by primitive procedures like `car` and `+`:

```
(+ (delay (* 3 7)) 13)          ⇒  34
```

10.6 Streams

In addition to promises, MIT/GNU Scheme supports a higher-level abstraction called *streams*. Streams are similar to lists, except that the tail of a stream is not computed until it is referred to. This allows streams to be used to represent infinitely long lists.

`stream` *object ...* [procedure]
Returns a newly allocated stream whose elements are the arguments. Note that the expression `(stream)` returns the empty stream, or end-of-stream marker.

`list->stream` *list* [procedure]
Returns a newly allocated stream whose elements are the elements of *list*. Equivalent to `(apply stream` *`list`*`)`.

`stream->list` *stream* [procedure]
Returns a newly allocated list whose elements are the elements of *stream*. If *stream* has infinite length this procedure will not terminate. This could have been defined by

```
(define (stream->list stream)
  (if (stream-null? stream)
      '()
      (cons (stream-car stream)
            (stream->list (stream-cdr stream)))))
```

`cons-stream` *object expression* [special form]
Returns a newly allocated stream pair. Equivalent to `(cons` *`object`* `(delay` *`expression`*`))`.

`stream-pair?` *object* [procedure]
Returns `#t` if *object* is a pair whose cdr contains a promise. Otherwise returns `#f`. This could have been defined by

```
(define (stream-pair? object)
  (and (pair? object)
       (promise? (cdr object))))
```

`stream-car` *stream* [procedure]
`stream-first` *stream* [procedure]
Returns the first element in *stream*. `stream-car` is equivalent to `car`. `stream-first` is a synonym for `stream-car`.

`stream-cdr` *stream* [procedure]
`stream-rest` *stream* [procedure]
Returns the first tail of *stream*. Equivalent to `(force (cdr stream))`. `stream-rest` is a synonym for `stream-cdr`.

`stream-null?` *stream* [procedure]
Returns `#t` if *stream* is the end-of-stream marker; otherwise returns `#f`. This is equivalent to `null?`, but should be used whenever testing for the end of a stream.

`stream-length` *stream* [procedure]
Returns the number of elements in *stream*. If *stream* has an infinite number of elements this procedure will not terminate. Note that this procedure forces all of the promises that comprise *stream*.

`stream-ref` *stream k* [procedure]
Returns the element of *stream* that is indexed by *k*; that is, the *k*th element. *K* must be an exact non-negative integer strictly less than the length of *stream*.

`stream-head` *stream k* [procedure]
Returns the first *k* elements of *stream* as a list. *K* must be an exact non-negative integer strictly less than the length of *stream*.

`stream-tail` *stream k* [procedure]
Returns the tail of *stream* that is indexed by *k*; that is, the *k*th tail. This is equivalent to performing `stream-cdr` *k* times. *K* must be an exact non-negative integer strictly less than the length of *stream*.

`stream-map` *procedure stream stream . . .* [procedure]
Returns a newly allocated stream, each element being the result of invoking *procedure* with the corresponding elements of the *stream*s as its arguments.

10.7 Weak References

Weak references are a mechanism for building data structures that point at objects without protecting them from garbage collection. An example of such a data structure might be an entry in a lookup table that should be removed if the rest of the program does not reference its key. Such an entry must still point at its key to carry out comparisons, but should not in itself prevent its key from being garbage collected.

A *weak reference* is a reference that points at an object without preventing it from being garbage collected. The term *strong reference* is used to distinguish normal references from weak ones. If there is no path of strong references to some object, the garbage collector will reclaim that object and mark any weak references to it to indicate that it has been reclaimed.

If there is a path of strong references from an object A to an object B, A is said to hold B *strongly*. If there is a path of references from an object A to an object B, but every such path traverses at least one weak reference, A is said to hold B *weakly*.

MIT Scheme provides two mechanisms for using weak references. *Weak pairs* are like normal pairs, except that their car slot is a weak reference (but the cdr is still strong). The heavier-weight *ephemerons* additionally arrange that the ephemeron does not count as holding the object in its key field strongly even if the object in its datum field does.

Warning: Working with weak references is subtle and requires careful analysis; most programs should avoid working with them directly. The most common use cases for weak references ought to be served by hash tables (see Section 11.4 [Hash Tables], page 185), which can employ various flavors of weak entry types, 1d tables (see Section 11.2 [1D Tables], page 183), which hold their keys weakly, and the association table (see Section 11.3 [The Association Table], page 184), which also holds its keys weakly.

10.7.1 Weak Pairs

The car of a *weak pair* holds its pointer weakly, while the cdr holds its pointer strongly. If the object in the car of a weak pair is not held strongly by any other data structure, it will be garbage-collected, and the original value replaced with a unique *reclaimed object*.

Note: weak pairs can be defeated by cross references among their slots. Consider a weak pair P holding an object A in its car and an object D in its cdr. P points to A weakly and to D strongly. If D holds A strongly, however, then P ends up holding A strongly after all. If avoiding this is worth a heavier-weight structure, See Section 10.7.2 [Ephemerons], page 179.

Note: weak pairs are *not* pairs; that is, they do not satisfy the predicate `pair?`.

`weak-pair?` *object* [procedure]
: Returns `#t` if *object* is a weak pair; otherwise returns `#f`.

`weak-cons` *car cdr* [procedure]
: Allocates and returns a new weak pair, with components *car* and *cdr*. The *car* component is held weakly.

`gc-reclaimed-object?` *object* [procedure]
: Returns `#t` if *object* is the reclaimed object, and `#f` otherwise.

`gc-reclaimed-object` [procedure]
: Returns the reclaimed object.

`weak-pair/car?` *weak-pair* [obsolete procedure]
: This predicate returns `#f` if the car of *weak-pair* has been garbage-collected; otherwise returns `#t`. In other words, it is true if *weak-pair* has a valid car component.

  This is equivalent to

  ```
  (not (gc-reclaimed-object? (weak-car weak-pair)))
  ```

  This predicate has been deprecated; instead use `gc-reclaimed-object?`. Please note that the previously recommended way to use `weak-pair/car?` will no longer work, so any code using it should be rewritten.

`weak-car` *weak-pair* [procedure]
Returns the car component of *weak-pair*. If the car component has been garbage-collected, this operation returns the reclaimed object.

`weak-set-car!` *weak-pair object* [procedure]
Sets the car component of *weak-pair* to *object* and returns an unspecified result.

`weak-cdr` *weak-pair* [procedure]
Returns the cdr component of *weak-pair*.

`weak-set-cdr!` *weak-pair object* [procedure]
Sets the cdr component of *weak-pair* to *object* and returns an unspecified result.

10.7.2 Ephemerons

An *ephemeron* is an object with two weakly referenced components called its *key* and *datum*. The garbage collector drops an ephemeron's references to both key and datum, rendering the ephemeron *broken*, if and only if the garbage collector can prove that there are no strong references to the key. In other words, an ephemeron is broken when nobody else cares about its key. In particular, the datum holding a reference to the key will not in itself prevent the ephemeron from becoming broken; in contrast, See Section 10.7.1 [Weak Pairs], page 178. Once broken, ephemerons never cease to be broken; setting the key or datum of a broken ephemeron with `set-ephemeron-key!` or `set-ephemeron-datum!` has no effect. Note that an ephemeron's reference to its datum may be dropped even if the datum is still reachable; all that matters is whether the key is reachable.

Ephemerons are considerably heavier-weight than weak pairs, because garbage-collecting ephemerons is more complicated than garbage-collecting weak pairs. Each ephemeron needs five words of storage, rather than the two words needed by a weak pair. However, while the garbage collector spends more time on ephemerons than on other objects, the amount of time it spends on ephemerons scales linearly with the number of live ephemerons, which is how its running time scales with the total number of live objects anyway.

`ephemeron?` *object* [procedure]
Returns `#t` if *object* is a ephemeron; otherwise returns `#f`.

`make-ephemeron` *key datum* [procedure]
Allocates and returns a new ephemeron, with components *key* and *datum*.

`ephemeron-broken?` *ephemeron* [procedure]
Returns `#t` if the garbage collector has dropped *ephemeron*'s references to its key and datum; otherwise returns `#f`.

`ephemeron-key` *ephemeron* [procedure]
`ephemeron-datum` *ephemeron* [procedure]
These return the key or datum component, respectively, of *ephemeron*. If *ephemeron* has been broken, these operations return `#f`, but they can also return `#f` if that is the value that was stored in the key or value component.

`set-ephemeron-key!` *ephemeron object* [procedure]
`set-ephemeron-datum!` *ephemeron object* [procedure]
These set the key or datum component, respectively, of *ephemeron* to *object* and return an unspecified result. If *ephemeron* is broken, neither of these operations has any effect.

Like `weak-pair/car?`, `ephemeron-broken?` must be used with care. If (`ephemeron-broken?` *ephemeron*) yields false, it guarantees only that prior evaluations of (`ephemeron-key` *ephemeron*) or (`ephemeron-datum` *ephemeron*) yielded the key or datum that was stored in the ephemeron, but it makes no guarantees about subsequent calls to `ephemeron-key` or `ephemeron-datum`: the garbage collector may run and break the ephemeron immediately after `ephemeron-broken?` returns. Thus, the correct idiom to fetch an ephemeron's key and datum and use them if the ephemeron is not broken is

```
(let ((key (ephemeron-key ephemeron))
      (datum (ephemeron-datum ephemeron)))
  (if (ephemeron-broken? ephemeron)
      ... broken case ...
      ... code using key and datum ...))
```

10.7.3 Reference barriers

The garbage collector may break an ephemeron if it can prove that the key is not strongly reachable. To ensure that it does not do so before a certain point in a program, the program can invoke a *reference barrier* on the key by calling the `reference-barrier` procedure, which guarantees that even if the program does not use the key, it will be considered strongly reachable until after `reference-barrier` returns.

`reference-barrier` *object* [procedure]
Guarantee that *object* is strongly reachable until after `reference-barrier` returns.

11 Associations

MIT/GNU Scheme provides several mechanisms for associating objects with one another. Each of these mechanisms creates a link between one or more objects, called *keys*, and some other object, called a *datum*. Beyond this common idea, however, each of the mechanisms has various different properties that make it appropriate in different situations:

- *Association lists* are one of Lisp's oldest association mechanisms. Because they are made from ordinary pairs, they are easy to build and manipulate, and very flexible in use. However, the average lookup time for an association list is linear in the number of associations.
- *1D tables* have a very simple interface, making them easy to use, and offer the feature that they do not prevent their keys from being reclaimed by the garbage collector. Like association lists, their average lookup time is linear in the number of associations; but 1D tables aren't as flexible.
- *The association table* is MIT/GNU Scheme's equivalent to the *property lists* of Lisp. It has the advantages that the keys may be any type of object and that it does not prevent the keys from being reclaimed by the garbage collector. However, two linear-time lookups must be performed, one for each key, whereas for traditional property lists only one lookup is required for both keys.
- *Hash tables* are a powerful mechanism with constant-time access to large amounts of data. Hash tables are not as flexible as association lists, but because their access times are independent of the number of associations in the table, for most applications they are the mechanism of choice.
- *Balanced binary trees* are another association mechanism that is useful for applications in which the keys are ordered. Binary trees have access times that are proportional to the logarithm of the number of associations in the tree. While they aren't as fast as hash tables, they offer the advantage that the contents of the tree can be converted to a sorted alist in linear time. Additionally, two trees can be compared for equality in worst-case linear time.
- *Red-Black trees* are a kind of balanced binary tree. The implementation supports destructive insertion and deletion operations with a good constant factor.
- *Weight-Balanced trees* are a kind of balanced binary tree. The implementation provides non-destructive operations. There is a comprehensive set of operations, including: a constant-time size operation; many high-level operations such as the set operations union, intersection and difference; and indexing of elements by position.

11.1 Association Lists

An *association list*, or *alist*, is a data structure used very frequently in Scheme. An alist is a list of pairs, each of which is called an *association*. The car of an association is called the *key*.

An advantage of the alist representation is that an alist can be incrementally augmented simply by adding new entries to the front. Moreover, because the searching procedures `assv` et al. search the alist in order, new entries can "shadow" old entries. If an alist is

viewed as a mapping from keys to data, then the mapping can be not only augmented but also altered in a non-destructive manner by adding new entries to the front of the alist.[1]

`alist?` *object* [procedure]

Returns `#t` if *object* is an association list (including the empty list); otherwise returns `#f`. Any *object* satisfying this predicate also satisfies `list?`.

`assq` *object alist* [procedure]
`assv` *object alist* [procedure]
`assoc` *object alist* [procedure]

These procedures find the first pair in *alist* whose car field is *object*, and return that pair; the returned pair is always an *element* of *alist*, *not* one of the pairs from which *alist* is composed. If no pair in *alist* has *object* as its car, `#f` (n.b.: not the empty list) is returned. `assq` uses `eq?` to compare *object* with the car fields of the pairs in *alist*, while `assv` uses `eqv?` and `assoc` uses `equal?`.[2]

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                              ⇒  (a 1)
(assq 'b e)                              ⇒  (b 2)
(assq 'd e)                              ⇒  #f
(assq (list 'a) '(((a)) ((b)) ((c))))   ⇒  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))  ⇒  ((a))
(assq 5 '((2 3) (5 7) (11 13)))          ⇒  unspecified
(assv 5 '((2 3) (5 7) (11 13)))          ⇒  (5 7)
```

`association-procedure` *predicate selector* [procedure]

Returns an association procedure that is similar to `assv`, except that *selector* (a procedure of one argument) is used to select the key from the association, and *predicate* (an equivalence predicate) is used to compare the key to the given item. This can be used to make association lists whose elements are, say, vectors instead of pairs (also see Section 7.6 [Searching Lists], page 149).

For example, here is how `assv` could be implemented:

```
(define assv (association-procedure eqv? car))
```

Another example is a "reverse association" procedure:

```
(define rassv (association-procedure eqv? cdr))
```

`del-assq` *object alist* [procedure]
`del-assv` *object alist* [procedure]
`del-assoc` *object alist* [procedure]

These procedures return a newly allocated copy of *alist* in which all associations with keys equal to *object* have been removed. Note that while the returned copy is a newly allocated list, the association pairs that are the elements of the list are shared with *alist*, not copied. `del-assq` uses `eq?` to compare *object* with the keys, while `del-assv` uses `eqv?` and `del-assoc` uses `equal?`.

---

[1] This introduction is taken from *Common Lisp, The Language*, second edition, p. 431.

[2] Although they are often used as predicates, `assq`, `assv`, and `assoc` do not have question marks in their names because they return useful values rather than just `#t` or `#f`.

```
(define a
  '((butcher . "231 e22nd St.")
    (baker . "515 w23rd St.")
    (hardware . "988 Lexington Ave.")))

(del-assq 'baker a)
     ⇒
     ((butcher . "231 e22nd St.")
      (hardware . "988 Lexington Ave."))
```

`del-assq!` *object alist* [procedure]
`del-assv!` *object alist* [procedure]
`del-assoc!` *object alist* [procedure]

These procedures remove from *alist* all associations with keys equal to *object*. They return the resulting list. `del-assq!` uses `eq?` to compare *object* with the keys, while `del-assv!` uses `eqv?` and `del-assoc!` uses `equal?`. These procedures are like `del-assq`, `del-assv`, and `del-assoc`, respectively, except that they destructively modify *alist*.

`delete-association-procedure` *deletor predicate selector* [procedure]

This returns a deletion procedure similar to `del-assv` or `del-assq!`. The *predicate* and *selector* arguments are the same as those for `association-procedure`, while the *deletor* argument should be either the procedure `list-deletor` (for non-destructive deletions), or the procedure `list-deletor!` (for destructive deletions).

For example, here is a possible implementation of `del-assv`:

```
(define del-assv
  (delete-association-procedure list-deletor eqv? car))
```

`alist-copy` *alist* [procedure]

Returns a newly allocated copy of *alist*. This is similar to `list-copy` except that the "association" pairs, i.e. the elements of the list *alist*, are also copied. `alist-copy` could have been implemented like this:

```
(define (alist-copy alist)
  (if (null? alist)
      '()
      (cons (cons (car (car alist)) (cdr (car alist)))
            (alist-copy (cdr alist)))))
```

11.2 1D Tables

1D tables ("one-dimensional" tables) are similar to association lists. In a 1D table, unlike an association list, the keys of the table are held *weakly*: if a key is garbage-collected, its associated value in the table is removed. 1D tables compare their keys for equality using `eq?`.

1D tables can often be used as a higher-performance alternative to the two-dimensional association table (see Section 11.3 [The Association Table], page 184). If one of the keys being associated is a compound object such as a vector, a 1D table can be stored in one

of the vector's slots. Under these circumstances, accessing items in a 1D table will be comparable in performance to using a property list in a conventional Lisp.

`make-1d-table` [procedure]
Returns a newly allocated empty 1D table.

`1d-table?` *object* [procedure]
Returns `#t` if *object* is a 1D table, otherwise returns `#f`. Any object that satisfies this predicate also satisfies `list?`.

`1d-table/put!` *1d-table key datum* [procedure]
Creates an association between *key* and *datum* in *1d-table*. Returns an unspecified value.

`1d-table/remove!` *1d-table key* [procedure]
Removes any association for *key* in *1d-table* and returns an unspecified value.

`1d-table/get` *1d-table key default* [procedure]
Returns the *datum* associated with *key* in *1d-table*. If there is no association for *key*, *default* is returned.

`1d-table/lookup` *1d-table key if-found if-not-found* [procedure]
If-found must be a procedure of one argument, and *if-not-found* must be a procedure of no arguments. If *1d-table* contains an association for *key*, *if-found* is invoked on the *datum* of the association. Otherwise, *if-not-found* is invoked with no arguments. In either case, the result of the invoked procedure is returned as the result of `1d-table/lookup`.

`1d-table/alist` *1d-table* [procedure]
Returns a newly allocated association list that contains the same information as *1d-table*.

11.3 The Association Table

MIT/GNU Scheme provides a generalization of the property-list mechanism found in most other implementations of Lisp: a global two-dimensional *association table*. This table is indexed by two keys, called *x-key* and *y-key* in the following procedure descriptions. These keys and the datum associated with them can be arbitrary objects. `eq?` is used to discriminate keys.

Think of the association table as a matrix: a single datum can be accessed using both keys, a column using *x-key* only, and a row using *y-key* only.

`2d-put!` *x-key y-key datum* [procedure]
Makes an entry in the association table that associates *datum* with *x-key* and *y-key*. Returns an unspecified result.

`2d-remove!` *x-key y-key* [procedure]
If the association table has an entry for *x-key* and *y-key*, it is removed. Returns an unspecified result.

`2d-get` *x-key y-key* [procedure]
Returns the *datum* associated with *x-key* and *y-key*. Returns `#f` if no such association exists.

`2d-get-alist-x` *x-key* [procedure]
Returns an association list of all entries in the association table that are associated with *x-key*. The result is a list of `(`*y-key* `.` *datum*`)` pairs. Returns the empty list if no entries for *x-key* exist.

```
(2d-put! 'foo 'bar 5)
(2d-put! 'foo 'baz 6)
(2d-get-alist-x 'foo)           ⇒  ((baz . 6) (bar . 5))
```

`2d-get-alist-y` *y-key* [procedure]
Returns an association list of all entries in the association table that are associated with *y-key*. The result is a list of `(`*x-key* `.` *datum*`)` pairs. Returns the empty list if no entries for *y-key* exist.

```
(2d-put! 'bar 'foo 5)
(2d-put! 'baz 'foo 6)
(2d-get-alist-y 'foo)           ⇒  ((baz . 6) (bar . 5))
```

11.4 Hash Tables

Hash tables are a fast, powerful mechanism for storing large numbers of associations. MIT/GNU Scheme's hash tables feature automatic resizing, customizable growth parameters, customizable hash procedures, and many options for weak references to keys or data.

The average times for the insertion, deletion, and lookup operations on a hash table are bounded by a constant. The space required by the table is proportional to the number of associations in the table; the constant of proportionality is described below (see Section 11.4.3 [Resizing of Hash Tables], page 193).

The hash table interface described below is a superset of SRFI 69: "Basic hash tables". The reason for supporting the extra functionality is that SRFI 69 fails to specify certain optimization-enabling exceptions to its semantics, forcing a correct implementation to pay the non-negligible performance cost of completely safe behavior.[3] The MIT/GNU Scheme native hash table interface, in contrast, specifies the minor exceptions it needs, and is therefore implemented more efficiently.

We do not describe the SRFI 69-compliant interface here, as that would be redundant with the SRFI document.

11.4.1 Construction of Hash Tables

The next few procedures are hash-table constructors. All hash table constructors are procedures that accept one optional argument, *initial-size*, and return a newly allocated hash

[3] SRFI 69 does not give hash functions the flexibility to return new hash values after a garbage collection, which prevents a system whose garbage collector may relocate objects from hashing based on the addresses of objects in memory (see Section 11.4.4 [Address Hashing], page 195). SRFI 69 also does not specify circumstances when procedures passed as arguments to hash table operations may not themselves modify the hash table, which requires defensive copying and defensive repetitions of lookups.

table. If *initial-size* is given, it must be an exact non-negative integer or `#f`. The meaning of *initial-size* is discussed below (see Section 11.4.3 [Resizing of Hash Tables], page 193).

Hash tables are normally characterized by two things: the equivalence predicate that is used to compare keys, and how the table allows its keys and data to be reclaimed by the garbage collector. If a table prevents its keys and data from being reclaimed by the garbage collector, it is said to hold its keys and data *strongly*; other arrangements are possible, where a table may hold keys or data *weakly* or *ephemerally* (see Section 10.7 [Weak References], page 177).

`make-strong-eq-hash-table` [*initial-size*] [procedure]
`make-symbol-hash-table` [*initial-size*] [obsolete procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eq?`. The keys and data are held strongly. These are the fastest of the standard hash tables.

`make-key-weak-eq-hash-table` [*initial-size*] [procedure]
`make-weak-eq-hash-table` [*initial-size*] [obsolete procedure]
`make-eq-hash-table` [*initial-size*] [obsolete procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eq?`. The keys are held weakly and the data are held strongly. Note that if a datum holds a key strongly, the table will effectively hold that key strongly.

`make-datum-weak-eq-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eq?`. The keys are held strongly and the data are held weakly. Note that if a key holds a datum strongly, the table will effectively hold that datum strongly.

`make-key-ephemeral-eq-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eq?`. The keys are held weakly, even if some of the data should hold some of the keys strongly.

`make-strong-eqv-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eqv?`. The keys and data are held strongly. These hash tables are a little slower than those made by `make-strong-eq-hash-table`.

`make-key-weak-eqv-hash-table` [*initial-size*] [procedure]
`make-weak-eqv-hash-table` [*initial-size*] [obsolete procedure]
`make-eqv-hash-table` [*initial-size*] [obsolete procedure]
`make-object-hash-table` [*initial-size*] [obsolete procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eqv?`. The keys are held weakly, except that booleans, characters, numbers, and interned symbols are held strongly. The data are held strongly. Note that if a datum holds a key strongly, the table will effectively hold that key strongly.

`make-datum-weak-eqv-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eqv?`. The keys are held strongly and the data are held weakly. Note that if a key holds a datum strongly, the table will effectively hold that datum strongly.

`make-key-ephemeral-eqv-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `eqv?`. The keys are held weakly, except that booleans, characters, numbers, and interned symbols are held strongly. The keys are effectively held weakly even if some of the data should hold some of the keys strongly.

`make-equal-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts arbitrary objects as keys, and compares those keys with `equal?`. The keys and data are held strongly. These hash tables are quite a bit slower than those made by `make-strong-eq-hash-table`.

`make-string-hash-table` [*initial-size*] [procedure]
Returns a newly allocated hash table that accepts character strings as keys, and compares them with `string=?`. The keys and data are held strongly.

All of the above are highly optimized table implementations. Next are some general constructors that allow for more flexible table definitions.

`make-hash-table` *comparator arg . . .* [procedure]
`make-hash-table` [*key=? [hash-function arg . . .]*] [procedure]
`alist->hash-table` *alist comparator arg . . .* [procedure]
`alist->hash-table` *alist* [*key=? [hash-function arg . . .]*] [procedure]
These are the standard constructors for making hash tables. The behavior of each differs depending on its arguments: if the first argument is a comparator, then it behaves like a SRFI 125 procedure, otherwise it behaves like a SRFI 69 procedure.

For SRFI 125 behavior the *comparator* must be a comparator that satisfies `comparator-hashable?`. The remaining *args* are optional, and may include the following symbols:

`weak-keys`
Specifies that the table will have weak keys.

`weak-values`
Specifies that the table will have weak values.

`ephemeral-keys`
Specifies that the table will have ephemeral keys.

`ephemeral-values`
Specifies that the table will have ephemeral values.

The symbols `weak-keys` and `weak-values` can be specified together or separately, likewise for `ephemeral-keys` and `ephemeral-values`. But weak and ephemeral symbols can't be mixed. If none of these symbols are present, then the keys and values are strongly held.

Additionally *args* may contain an exact non-negative integer, which specifies an initial size for the table; otherwise a default size is used.

For SRFI 69 behavior the *key=?* argument specifies how keys are compared and defaults to `equal?`. The *hash-function* argument specifies the hash function to use. If *hash-function* is not specified, it defaults to a standard value that depends on *key=?*; an error is signaled if there's no standard value. The *arg* arguments are allowed but are implementation dependent; do not provide them.

The procedure `alist->hash-table` creates a new hash table, as with `make-hash-table`, and then fills it with the contents of *alist*.

The remaining constructors use *hash-table types* to encapsulate the hashing parameters.

`make-hash-table*` *type* [*initial-size*] [obsolete procedure]
Constructs a new hash table using the hashing parameters in *type*.

`hash-table-constructor` *comparator arg ...* [procedure]
`hash-table-constructor` *type* [obsolete procedure]
Returns a procedure that, when called, constructs a new hash table using the specified parameters. The returned procedure accepts an optional *initial-size*.

If its first argument is a comparator, it uses the *comparator* and *args* as in `make-hash-table`, except that any initial size specified in *args* can be overridden by the *initial-size* argument to the returned procedure.

If its first argument is a type, returns a procedure that, when called, constructs a new hash table using the hashing parameters in *type*. This is equivalent to

```
(lambda (#!optional initial-size)
  (make-hash-table* type initial-size))
```

The next two procedures are used to create hash-table types. The procedures are equivalent in power; they differ only in how the types are described.

`make-hash-table-type` *hash-function key=? rehash-after-gc? entry-type* [obsolete procedure]
This procedure accepts four arguments and returns a hash-table type, which can be used to make hash tables of that type. The *key=?* argument is an equivalence predicate for the keys of the hash table. The *hash-function* argument is a procedure that computes a hash number. Specifically, *hash-function* accepts two arguments, a key and an exact positive integer (the *modulus*), and returns an exact non-negative integer that is less than the modulus.

The argument *rehash-after-gc?*, if true, says that the values returned by *hash-function* might change after a garbage collection. If so, the hash-table implementation arranges for the table to be rehashed when necessary. (See Section 11.4.4 [Address Hashing], page 195, for information about hash procedures that have this property.) Otherwise, it is assumed that *hash-function* always returns the same value for the same arguments.

The argument *entry-type* determines the strength with which the hash table will hold its keys and values. It must be one of the entry-type variables described below, which all start with `hash-table-entry-type:`.

`make-hash-table-type*` *key=? hash-function rehash-after-gc? entry-type* [obsolete procedure]

This procedure's arguments, except for *key=?*, are keyword arguments; that is, each argument is a symbol of the same name followed by its value. Aside from how they are passed, the arguments have the same meaning as those for `make-hash-table-type`. Note that all of the keyword arguments are optional, while *key=?* is required.

The argument *entry-type* specifies *the name of* an entry type. It must be a symbol corresponding to one of the entry-type variables described below. The name of an entry type is the symbol composed of the suffix of the corresponding variable; for example the type `hash-table-entry-type:key-weak` has the name `key-weak`.

The default values for the keyword arguments are as follows. The arguments *hash-function* and *rehash-after-gc?* default to standard values that depend on *key=?*; an error is signaled if *key=?* has no standard values. The argument *entry-type* defaults to *strong*.

`hash-table-entry-type:strong` [obsolete variable]

The entry type for hash tables that hold both keys and data strongly.

`hash-table-entry-type:key-weak` [obsolete variable]

An entry type for hash tables that hold keys weakly and data strongly. An entry of this type is a weak pair (see Section 10.7.1 [Weak Pairs], page 178) whose weak (car) slot holds the key of the entry and whose strong (cdr) slot holds the datum of the entry. If a key of such a hash table is garbage collected, the corresponding entry will be removed. Note that if some datum holds some key strongly, the table will effectively hold that key strongly.

`hash-table-entry-type:datum-weak` [obsolete variable]

An entry type for hash tables that hold keys strongly and data weakly. An entry of this type is a weak pair (see Section 10.7.1 [Weak Pairs], page 178) whose weak (car) slot holds the datum of the entry and whose strong (cdr) slot holds the key of the entry. If a datum of such a hash table is garbage collected, all corresponding entries will be removed. Note that if some key holds some datum strongly, the table will effectively hold that datum strongly.

`hash-table-entry-type:key&datum-weak` [obsolete variable]
`hash-table-entry-type:key/datum-weak` [obsolete variable]

The entry type for hash tables that hold both keys and data weakly. An entry of this type is a weak list, holding both the key and the datum in the weak (car) slot of weak pairs (see Section 10.7.1 [Weak Pairs], page 178). If either a key or datum of such a hash table is garbage collected, all corresponding entries will be removed.

`hash-table-entry-type:key-ephemeral` [obsolete variable]

An entry type for hash tables that hold data ephemerally, keyed by the keys. An entry of this type is an ephemeron (see Section 10.7.2 [Ephemerons], page 179) whose key is the key of the entry and whose datum is the datum of the entry. If a key of such a hash table is garbage collected, the corresponding entry will be removed. Note that the table holds all its keys weakly even if some data should hold some keys strongly.

`hash-table-entry-type:datum-ephemeral` [obsolete variable]
: An entry type for hash tables that hold keys ephemerally, keyed by the data. An entry of this type is an ephemeron (see Section 10.7.2 [Ephemerons], page 179) whose key is the datum of the entry and whose datum is the key of the entry. If a datum of such a hash table is garbage collected, all corresponding entries will be removed. Note that the table holds all its data weakly even if some keys should hold some data strongly.

`hash-table-entry-type:key&datum-ephemeral` [obsolete variable]
: The entry type for hash tables that hold both keys and data ephemerally keyed on each other. An entry of this type is a pair of ephemerons (see Section 10.7.2 [Ephemerons], page 179), one holding the datum keyed by the key and the other holding the key keyed by the datum. If both the key and the datum of any entry of such a hash table are garbage collected, the entry will be removed. The table holds all its keys and data weakly itself, but will prevent any key or datum from being garbage collected if there are strong references to its datum or key, respectively.

Some examples showing how some standard hash-table constructors could have been defined:

```
(define make-weak-eq-hash-table
  (hash-table-constructor
    (make-hash-table-type eq-hash eq? #t
      hash-table-entry-type:key-weak)))

(define make-equal-hash-table
  (hash-table-constructor
    (make-hash-table-type equal-hash equal? #t
      hash-table-entry-type:strong)))

(define make-string-hash-table
  (hash-table-constructor
    (make-hash-table-type string-hash string=? #f
      hash-table-entry-type:strong)))
```

The following procedures are provided only for backward compatibility. They should be considered **deprecated** and should not be used in new programs.

`hash-table/constructor` *hash-function key=? rehash-after-gc? entry-type* [obsolete procedure]
: This procedure is **deprecated**. Instead use the equivalent

```
(hash-table-constructor
  (make-hash-table-type hash-function key=? rehash-after-gc?
                        entry-type))
```

`strong-hash-table/constructor` *hash-function key=?* [*rehash-after-gc?*] [obsolete procedure]
: Like `hash-table/constructor` but always uses `hash-table-entry-type:strong`. If *rehash-after-gc?* is omitted, it defaults to `#f`.

`weak-hash-table/constructor` *hash-function key=? [rehash-after-gc?]* [obsolete procedure]

Like `hash-table/constructor` but always uses `hash-table-entry-type:key-weak`. If *rehash-after-gc?* is omitted, it defaults to `#f`.

11.4.2 Basic Hash Table Operations

The procedures described in this section are the basic operations on hash tables. They provide the functionality most often needed by programmers. Subsequent sections describe other operations that provide additional functionality needed by some applications.

`hash-table?` *object* [procedure]

Returns `#t` if *object* is a hash table, otherwise returns `#f`.

`hash-table-set!` *hash-table key datum* [procedure]
`hash-table/put!` *hash-table key datum* [obsolete procedure]

Associates *datum* with *key* in *hash-table* and returns an unspecified result.

The average time required by this operation is bounded by a constant.

`hash-table-ref` *hash-table key* [*get-default*] [procedure]

Returns the datum associated with *key* in *hash-table*. If there is no association for *key*, and *get-default* is provided, it is called with no arguments and the value it yields is returned; if *get-default* is not provided, an error is signaled.

The average time required by this operation is bounded by a constant.

`hash-table-ref/default` *hash-table key default* [procedure]
`hash-table/get` *hash-table key default* [obsolete procedure]

Equivalent to

```
(hash-table-ref hash-table key (lambda () default))
```

`hash-table-delete!` *hash-table key* [procedure]
`hash-table/remove!` *hash-table key* [obsolete procedure]

If *hash-table* has an association for *key*, removes it. Returns an unspecified result.

The average time required by this operation is bounded by a constant.

`hash-table-clear!` *hash-table* [procedure]
`hash-table/clear!` *hash-table* [obsolete procedure]

Removes all associations in *hash-table* and returns an unspecified result.

The average and worst-case times required by this operation are bounded by a constant.

`hash-table-size` *hash-table* [procedure]
`hash-table/count` *hash-table* [obsolete procedure]

Returns the number of associations in *hash-table* as an exact non-negative integer. If *hash-table* does not hold its keys and data strongly, this is a conservative upper bound that may count some associations whose keys or data have recently been reclaimed by the garbage collector.

The average and worst-case times required by this operation are bounded by a constant.

`hash-table->alist` *hash-table* [procedure]
Returns the contents of *hash-table* as a newly allocated alist. Each element of the alist is a pair `(`*key* `.` *datum*`)` where *key* is one of the keys of *hash-table*, and *datum* is its associated datum.

The average and worst-case times required by this operation are linear in the number of associations in the table.

`hash-table-keys` *hash-table* [procedure]
`hash-table/key-list` *hash-table* [obsolete procedure]
Returns a newly allocated list of the keys in *hash-table*.

The average and worst-case times required by this operation are linear in the number of associations in the table.

`hash-table-values` *hash-table* [procedure]
`hash-table/datum-list` *hash-table* [obsolete procedure]
Returns a newly allocated list of the datums in *hash-table*. Each element of the list corresponds to one of the associations in *hash-table*; if the table contains multiple associations with the same datum, so will this list.

The average and worst-case times required by this operation are linear in the number of associations in the table.

`hash-table-walk` *hash-table procedure* [procedure]
`hash-table/for-each` *hash-table procedure* [obsolete procedure]
Procedure must be a procedure of two arguments. Invokes *procedure* once for each association in *hash-table*, passing the association's *key* and *datum* as arguments, in that order. Returns an unspecified result. *Procedure* must not modify *hash-table*, with one exception: it is permitted to call `hash-table-delete!` to remove the association being processed.

The following procedure is useful when there is no sensible default value for `hash-table-ref` and the caller must choose between different actions depending on whether there is a datum associated with the key.

`hash-table/lookup` *hash-table key if-found if-not-found* [obsolete procedure]
If-found must be a procedure of one argument, and *if-not-found* must be a procedure of no arguments. If *hash-table* contains an association for *key*, *if-found* is invoked on the datum of the association. Otherwise, *if-not-found* is invoked with no arguments. In either case, the result yielded by the invoked procedure is returned as the result of `hash-table/lookup` (`hash-table/lookup` *reduces* into the invoked procedure, i.e. calls it tail-recursively).

The average time required by this operation is bounded by a constant.

`hash-table-update!` *hash-table key procedure* [*get-default*] [procedure]
Procedure must be a procedure of one argument and *get-default*, if supplied, must be a procedure of zero arguments. Applies *procedure* to the datum associated with *key* in *hash-table* or to the value of calling *get-default* if there is no association for *key*, associates the result with *key*, and returns an unspecified value. If *get-default* is not supplied and there's no association for *key*, an error is signaled.

The average time required by this operation is bounded by a constant.

`hash-table-update!/default` *hash-table key procedure default* [procedure]
`hash-table/modify!` *hash-table key default procedure* [obsolete procedure]
Equivalent to

```
(hash-table-update! hash-table key procedure (lambda () default))
```

`hash-table-intern!` *hash-table key get-default* [procedure]
`hash-table/intern!` *hash-table key get-default* [obsolete procedure]
Get-default must be a procedure of zero arguments. Ensures that *hash-table* has an association for *key* and returns the associated datum. If *hash-table* did not have a datum associated with *key*, *get-default* is called and its value is used to create a new association for *key*.

The average time required by this operation is bounded by a constant.

The following procedure is sometimes useful in conjunction with weak and ephemeral hash tables. Normally it is not needed, because such hash tables clean themselves automatically as they are used.

`hash-table-clean!` *hash-table* [procedure]
`hash-table/clean!` *hash-table* [obsolete procedure]
If *hash-table* is a type of hash table that holds its keys or data weakly or ephemerally, this procedure recovers any space that was being used to record associations for objects that have been reclaimed by the garbage collector. Otherwise, this procedure does nothing. In either case, it returns an unspecified result.

11.4.3 Resizing of Hash Tables

Normally, hash tables automatically resize themselves according to need. Because of this, the programmer need not be concerned with management of the table's size. However, some limited control over the table's size is provided, which will be discussed below. This discussion involves two concepts, *usable size* and *physical size*, which we will now define.

The *usable size* of a hash table is the number of associations that the table can hold at a given time. If the number of associations in the table exceeds the usable size, the table will automatically grow, increasing the usable size to a new value that is sufficient to hold the associations.

The *physical size* is an abstract measure of a hash table that specifies how much space is allocated to hold the associations of the table. The physical size is always greater than or equal to the usable size. The physical size is not interesting in itself; it is interesting only for its effect on the performance of the hash table. While the average performance of a hash-table lookup is bounded by a constant, the worst-case performance is not. For a table containing a given number of associations, increasing the physical size of the table decreases the probability that worse-than-average performance will occur.

The physical size of a hash table is statistically related to the number of associations. However, it is possible to place bounds on the physical size, and from this to estimate the amount of space used by the table:

```
(define (hash-table-space-bounds count rehash-size rehash-threshold)
  (let ((tf (/ 1 rehash-threshold)))
    (values (if (exact-integer? rehash-size)
                (- (* count (+ 4 tf))
                   (* tf (+ rehash-size rehash-size)))
                (* count (+ 4 (/ tf (* rehash-size rehash-size)))))
            (* count (+ 4 tf)))))
```

What this formula shows is that, for a "normal" rehash size (that is, not an exact integer), the amount of space used by the hash table is proportional to the number of associations in the table. The constant of proportionality varies statistically, with the low bound being

```
(+ 4 (/ (/ 1 rehash-threshold) (* rehash-size rehash-size)))
```

and the high bound being

```
(+ 4 (/ 1 rehash-threshold))
```

which, for the default values of these parameters, are `4.25` and `5`, respectively. Reducing the rehash size will tighten these bounds, but increases the amount of time spent resizing, so you can see that the rehash size gives some control over the time-space tradeoff of the table.

The programmer can control the size of a hash table by means of three parameters:

- Each table's *initial-size* may be specified when the table is created.
- Each table has a *rehash size* that specifies how the size of the table is changed when it is necessary to grow or shrink the table.
- Each table has a *rehash threshold* that specifies the relationship of the table's physical size to its usable size.

If the programmer knows that the table will initially contain a specific number of items, *initial-size* can be given when the table is created. If *initial-size* is an exact non-negative integer, it specifies the initial usable size of the hash table; the table will not change size until the number of items in the table exceeds *initial-size*, after which automatic resizing is enabled and *initial-size* no longer has any effect. Otherwise, if *initial-size* is not given or is `#f`, the table is initialized to an unspecified size and automatic resizing is immediately enabled.

The *rehash size* specifies how much to increase the usable size of the hash table when it becomes full. It is either an exact positive integer, or a real number greater than one. If it is an integer, the new size is the sum of the old size and the rehash size. Otherwise, it is a real number, and the new size is the product of the old size and the rehash size. Increasing the rehash size decreases the average cost of an insertion, but increases the average amount of space used by the table. The rehash size of a table may be altered dynamically by the application in order to optimize the resizing of the table; for example, if the table will grow quickly for a known period and afterwards will not change size, performance might be improved by using a large rehash size during the growth phase and a small one during the static phase. The default rehash size of a newly constructed hash table is `2.0`.

Warning: The use of an exact positive integer for a rehash size is almost always undesirable; this option is provided solely for compatibility with the Common Lisp hash-table mechanism. The reason for this has to do with the time penalty for resizing the hash table. The time needed to resize a hash table is proportional to the number of associations in

the table. This resizing cost is *amortized* across the insertions required to fill the table to the point where it needs to grow again. If the table grows by an amount proportional to the number of associations, then the cost of resizing and the increase in size are both proportional to the number of associations, so the *amortized cost* of an insertion operation is still bounded by a constant. However, if the table grows by a constant amount, this is not true: the amortized cost of an insertion is not bounded by a constant. Thus, using a constant rehash size means that the average cost of an insertion increases proportionally to the number of associations in the hash table.

The *rehash threshold* is a real number, between zero exclusive and one inclusive, that specifies the ratio between a hash table's usable size and its physical size. Decreasing the rehash threshold decreases the probability of worse-than-average insertion, deletion, and lookup times, but increases the physical size of the table for a given usable size. The default rehash threshold of a newly constructed hash table is `1`.

`hash-table-grow-size` *hash-table* [procedure]
`hash-table/size` *hash-table* [obsolete procedure]
Returns the usable size of *hash-table* as an exact positive integer. This is the maximum number of associations that *hash-table* can hold before it will grow.

`hash-table-shrink-size` *hash-table* [procedure]
Returns the minimum number of associations that *hash-table* can hold before it will shrink.

`hash-table-rehash-size` *hash-table* [procedure]
`hash-table/rehash-size` *hash-table* [obsolete procedure]
Returns the rehash size of *hash-table*.

`set-hash-table-rehash-size!` *hash-table x* [procedure]
`set-hash-table/rehash-size!` *hash-table x* [obsolete procedure]
X must be either an exact positive integer, or a real number that is greater than one. Sets the rehash size of *hash-table* to *x* and returns an unspecified result. This operation adjusts the "shrink threshold" of the table; the table might shrink if the number of associations is less than the new threshold.

`hash-table-rehash-threshold` *hash-table* [procedure]
`hash-table/rehash-threshold` *hash-table* [obsolete procedure]
Returns the rehash threshold of *hash-table*.

`set-hash-table-rehash-threshold!` *hash-table x* [procedure]
`set-hash-table/rehash-threshold!` *hash-table x* [obsolete procedure]
X must be a real number between zero exclusive and one inclusive. Sets the rehash threshold of *hash-table* to *x* and returns an unspecified result. This operation does not change the usable size of the table, but it usually changes the physical size of the table, which causes the table to be rehashed.

11.4.4 Address Hashing

The procedures described in this section may be used to make very efficient key-hashing procedures for arbitrary objects. All of these procedures are based on *address hashing*,

which uses the address of an object as its hash number. The great advantage of address hashing is that converting an arbitrary object to a hash number is extremely fast and takes the same amount of time for any object.

The disadvantage of address hashing is that the garbage collector changes the addresses of most objects. The hash-table implementation compensates for this disadvantage by automatically rehashing tables that use address hashing when garbage collections occur. Thus, in order to use these procedures for key hashing, it is necessary to tell the hash-table implementation (by means of the *rehash-after-gc?* argument to the hash-table type constructors) that the hash numbers computed by your key-hashing procedure must be recomputed after a garbage collection.

`eq-hash` *object* [procedure]
`eqv-hash` *object* [procedure]
`equal-hash` *object* [procedure]
These procedures return a hash number for *object*. The result is always a non-negative integer, and in the case of `eq-hash`, a non-negative fixnum. Two objects that are equivalent according to `eq?`, `eqv?`, or `equal?`, respectively, will produce the same hash number when passed as arguments to these procedures, provided that the garbage collector does not run during or between the two calls.

`hash-by-identity` *key* [*modulus*] [procedure]
This SRFI 69 procedure returns the same value as `eq-hash`, optionally limited by *modulus*.

`hash` *key* [*modulus*] [procedure]
This SRFI 69 procedure returns the same value as `equal-hash`, optionally limited by *modulus*.

`hash-by-eqv` *key* [*modulus*] [obsolete procedure]
This procedure returns the same value as `eqv-hash`, optionally limited by *modulus*.

`eq-hash-mod` *object modulus* [obsolete procedure]
This procedure is the key-hashing procedure used by `make-strong-eq-hash-table`.

`eqv-hash-mod` *object modulus* [obsolete procedure]
This procedure is the key-hashing procedure used by `make-strong-eqv-hash-table`.

`equal-hash-mod` *object modulus* [obsolete procedure]
This procedure is the key-hashing procedure used by `make-equal-hash-table`.

11.5 Object Hashing

The MIT/GNU Scheme object-hashing facility provides a mechanism for generating a unique hash number for an arbitrary object. This hash number, unlike an object's address, is unchanged by garbage collection. The object-hashing facility is used in the generation of the written representation for many objects (see Section 14.10 [Custom Output], page 246), but it can be used for anything that needs a stable identifier for an arbitrary object.

All of these procedures accept an optional argument called *hasher* which contains the object-integer associations. If given, this argument must be an object hasher as constructed by `make-object-hasher` (see below). If not given, a default hasher is used.

`hash-object` *object* [*hasher*] [procedure]
`hash` *object* [*hasher*] [obsolete procedure]
`object-hash` *object* [*hasher*] [obsolete procedure]
`hash-object` associates an exact non-negative integer with *object* and returns that integer. If `hash-object` was previously called with *object* as its argument, the integer returned is the same as was returned by the previous call. `hash-object` guarantees that distinct objects (in the sense of `eqv?`) are associated with distinct integers.

`unhash-object` *k* [*hasher*] [procedure]
`unhash` *k* [*hasher*] [obsolete procedure]
`object-unhash` *k* [*hasher*] [obsolete procedure]
`unhash-object` takes an exact non-negative integer *k* and returns the object associated with that integer. If there is no object associated with *k*, or if the object previously associated with *k* has been reclaimed by the garbage collector, an error of type `condition-type:bad-range-argument` is signalled. In other words, if `hash-object` previously returned *k* for some object, and that object has not been reclaimed, it is the value of the call to `unhash-object`.

An object that is passed to `hash-object` as an argument is not protected from being reclaimed by the garbage collector. If all other references to that object are eliminated, the object will be reclaimed. Subsequently calling `unhash-object` with the hash number of the (now reclaimed) object will signal an error.

```
(define x (cons 0 0))          ⇒  unspecified
(hash-object x)                ⇒  77
(eqv? (hash-object x)
      (hash-object x))         ⇒  #t
(define x 0)                   ⇒  unspecified
(gc-flip)                      ;force a garbage collection
(unhash-object 77)             error
```

`object-hashed?` *object* [*hasher*] [procedure]
This predicate is true iff *object* has an associated hash number.

`valid-object-hash?` *k* [*hasher*] [procedure]
`valid-hash-number?` *k* [*hasher*] [obsolete procedure]
This predicate is true iff *k* is the hash number associated with some object.

Finally, this procedure makes new object hashers:

`make-object-hasher` [procedure]
`hash-table/make` [obsolete procedure]
This procedure creates and returns a new, empty object hasher that is suitable for use as the optional *hasher* argument to the above procedures. The returned hasher contains no associations.

11.6 Red-Black Trees

Balanced binary trees are a useful data structure for maintaining large sets of associations whose keys are ordered. While most applications involving large association sets should use

hash tables, some applications can benefit from the use of binary trees. Binary trees have two advantages over hash tables:

- The contents of a binary tree can be converted to an alist, sorted by key, in time proportional to the number of associations in the tree. A hash table can be converted into an unsorted alist in linear time; sorting it requires additional time.
- Two binary trees can be compared for equality in linear time. Hash tables, on the other hand, cannot be compared at all; they must be converted to alists before comparison can be done, and alist comparison is quadratic unless the alists are sorted.

MIT/GNU Scheme provides an implementation of *red-black* trees. The red-black tree-balancing algorithm provides generally good performance because it doesn't try to keep the tree very closely balanced. At any given node in the tree, one side of the node can be twice as high as the other in the worst case. With typical data the tree will remain fairly well balanced anyway.

A red-black tree takes space that is proportional to the number of associations in the tree. For the current implementation, the constant of proportionality is eight words per association.

Red-black trees hold their keys *strongly*. In other words, if a red-black tree contains an association for a given key, that key cannot be reclaimed by the garbage collector.

`make-rb-tree` *key=? key<?* [procedure]
: This procedure creates and returns a newly allocated red-black tree. The tree contains no associations. *Key=?* and *key<?* are predicates that compare two keys and determine whether they are equal to or less than one another, respectively. For any two keys, at most one of these predicates is true.

`rb-tree?` *object* [procedure]
: Returns `#t` if *object* is a red-black tree, otherwise returns `#f`.

`rb-tree/insert!` *rb-tree key datum* [procedure]
: Associates *datum* with *key* in *rb-tree* and returns an unspecified value. If *rb-tree* already has an association for *key*, that association is replaced. The average and worst-case times required by this operation are proportional to the logarithm of the number of assocations in *rb-tree*.

`rb-tree/lookup` *rb-tree key default* [procedure]
: Returns the datum associated with *key* in *rb-tree*. If *rb-tree* doesn't contain an association for *key*, *default* is returned. The average and worst-case times required by this operation are proportional to the logarithm of the number of assocations in *rb-tree*.

`rb-tree/ref` *rb-tree key* [*fail* [*succeed*]] [procedure]
: Returns the datum associated with *key* in *rb-tree*. If *rb-tree* doesn't contain an association for *key*, calls *fail* with argument *key* and returns that result; in this situation it is an error if *fail* is not provided. If there is an associated datum and *succeed* is provided, then it is called with that datum and the resulting value returned.

`rb-tree/delete!` *rb-tree key* [procedure]
If *rb-tree* contains an association for *key*, removes it. Returns an unspecified value. The average and worst-case times required by this operation are proportional to the logarithm of the number of assocations in *rb-tree*.

`rb-tree/update!` *rb-tree key updater* [*fail* [*succeed*]] [procedure]
Updates the datum associated with *key* in *rb-tree*. Calls *updater* with the current datum, and replaces that datum with the result of the call. If there is no current datum, calls *fail* with argument *key* and uses the resulting value in its place; in this situation it is an error if *fail* is not provided. If *succeed* is provided and there is an existing datum, then it is called with that datum and the resulting value is used in its place.

`rb-tree/intern!` *rb-tree key fail* [procedure]
Guarantees that *rb-tree* contains an association for *key*. If there is none, then *fail* is called with no arguments and the resulting value is associated with *key*.

`rb-tree->alist` *rb-tree* [procedure]
Returns the contents of *rb-tree* as a newly allocated alist. Each element of the alist is a pair `(`*key* `.` *datum*`)` where *key* is one of the keys of *rb-tree*, and *datum* is its associated datum. The alist is sorted by key according to the *key*`<?` argument used to construct *rb-tree*. The time required by this operation is proportional to the number of associations in the tree.

`rb-tree/key-list` *rb-tree* [procedure]
Returns a newly allocated list of the keys in *rb-tree*. The list is sorted by key according to the *key*`<?` argument used to construct *rb-tree*. The time required by this operation is proportional to the number of associations in the tree.

`rb-tree/datum-list` *rb-tree* [procedure]
Returns a newly allocated list of the datums in *rb-tree*. Each element of the list corresponds to one of the associations in *rb-tree*, so if the tree contains multiple associations with the same datum, so will this list. The list is sorted by the keys of the associations, even though they do not appear in the result. The time required by this operation is proportional to the number of associations in the tree.

This procedure is equivalent to:

```
(lambda (rb-tree) (map cdr (rb-tree->alist rb-tree)))
```

`rb-tree/equal?` *rb-tree-1 rb-tree-2 datum=?* [procedure]
Compares *rb-tree-1* and *rb-tree-2* for equality, returning `#t` iff they are equal and `#f` otherwise. The trees must have been constructed with the same equality and order predicates (same in the sense of `eq?`). The keys of the trees are compared using the *key=?* predicate used to build the trees, while the datums of the trees are compared using the equivalence predicate *datum=?*. The worst-case time required by this operation is proportional to the number of associations in the tree.

`rb-tree/empty?` *rb-tree* [procedure]
Returns `#t` iff *rb-tree* contains no associations. Otherwise returns `#f`.

`rb-tree/size` *rb-tree* [procedure]
Returns the number of associations in *rb-tree*, an exact non-negative integer. The average and worst-case times required by this operation are proportional to the number of associations in the tree.

`rb-tree/height` *rb-tree* [procedure]
Returns the height of *rb-tree*, an exact non-negative integer. This is the length of the longest path from a leaf of the tree to the root. The average and worst-case times required by this operation are proportional to the number of associations in the tree.

The returned value satisfies the following:

```
(lambda (rb-tree)
  (let ((size (rb-tree/size rb-tree))
        (lg (lambda (x) (/ (log x) (log 2)))))
    (<= (lg size)
        (rb-tree/height rb-tree)
        (* 2 (lg (+ size 1))))))
```

`rb-tree/copy` *rb-tree* [procedure]
Returns a newly allocated copy of *rb-tree*. The copy is identical to *rb-tree* in all respects, except that changes to *rb-tree* do not affect the copy, and vice versa. The time required by this operation is proportional to the number of associations in the tree.

`alist->rb-tree` *alist key=? key<?* [procedure]
Returns a newly allocated red-black tree that contains the same associations as *alist*. This procedure is equivalent to:

```
(lambda (alist key=? key<?)
  (let ((tree (make-rb-tree key=? key<?)))
    (for-each (lambda (association)
                (rb-tree/insert! tree
                                 (car association)
                                 (cdr association)))
              alist)
    tree))
```

The following operations provide access to the smallest and largest members in a red/black tree. They are useful for implementing priority queues.

`rb-tree/min` *rb-tree default* [procedure]
Returns the smallest key in *rb-tree*, or *default* if the tree is empty.

`rb-tree/min-datum` *rb-tree default* [procedure]
Returns the datum associated with the smallest key in *rb-tree*, or *default* if the tree is empty.

`rb-tree/min-pair` *rb-tree* [procedure]
Finds the smallest key in *rb-tree* and returns a pair containing that key and its associated datum. If the tree is empty, returns `#f`.

`rb-tree/max` *rb-tree default* [procedure]
Returns the largest key in *rb-tree*, or *default* if the tree is empty.

`rb-tree/max-datum` *rb-tree default* [procedure]
Returns the datum associated with the largest key in *rb-tree*, or *default* if the tree is empty.

`rb-tree/max-pair` *rb-tree* [procedure]
Finds the largest key in *rb-tree* and returns a pair containing that key and its associated datum. If the tree is empty, returns `#f`.

`rb-tree/delete-min!` *rb-tree default* [procedure]
`rb-tree/delete-min-datum!` *rb-tree default* [procedure]
`rb-tree/delete-min-pair!` *rb-tree* [procedure]
`rb-tree/delete-max!` *rb-tree default* [procedure]
`rb-tree/delete-max-datum!` *rb-tree default* [procedure]
`rb-tree/delete-max-pair!` *rb-tree* [procedure]
These operations are exactly like the accessors above, in that they return information associated with the smallest or largest key, except that they simultaneously delete that key.

11.7 Weight-Balanced Trees

Balanced binary trees are a useful data structure for maintaining large sets of ordered objects or sets of associations whose keys are ordered. MIT/GNU Scheme has a comprehensive implementation of weight-balanced binary trees which has several advantages over the other data structures for large aggregates:

- In addition to the usual element-level operations like insertion, deletion and lookup, there is a full complement of collection-level operations, like set intersection, set union and subset test, all of which are implemented with good orders of growth in time and space. This makes weight-balanced trees ideal for rapid prototyping of functionally derived specifications.
- An element in a tree may be indexed by its position under the ordering of the keys, and the ordinal position of an element may be determined, both with reasonable efficiency.
- Operations to find and remove minimum element make weight-balanced trees simple to use for priority queues.
- The implementation is *functional* rather than *imperative*. This means that operations like 'inserting' an association in a tree do not destroy the old tree, in much the same way that `(+ 1 x)` modifies neither the constant 1 nor the value bound to `x`. The trees are referentially transparent thus the programmer need not worry about copying the trees. Referential transparency allows space efficiency to be achieved by sharing subtrees.

These features make weight-balanced trees suitable for a wide range of applications, especially those that require large numbers of sets or discrete maps. Applications that have a few global databases and/or concentrate on element-level operations like insertion and lookup are probably better off using hash tables or red-black trees.

The *size* of a tree is the number of associations that it contains. Weight-balanced binary trees are balanced to keep the sizes of the subtrees of each node within a constant

factor of each other. This ensures logarithmic times for single-path operations (like lookup and insertion). A weight-balanced tree takes space that is proportional to the number of associations in the tree. For the current implementation, the constant of proportionality is six words per association.

Weight-balanced trees can be used as an implementation for either discrete sets or discrete maps (associations). Sets are implemented by ignoring the datum that is associated with the key. Under this scheme if an association exists in the tree this indicates that the key of the association is a member of the set. Typically a value such as `()`, `#t` or `#f` is associated with the key.

Many operations can be viewed as computing a result that, depending on whether the tree arguments are thought of as sets or maps, is known by two different names. An example is `wt-tree/member?`, which, when regarding the tree argument as a set, computes the set membership operation, but, when regarding the tree as a discrete map, `wt-tree/member?` is the predicate testing if the map is defined at an element in its domain. Most names in this package have been chosen based on interpreting the trees as sets, hence the name `wt-tree/member?` rather than `wt-tree/defined-at?`.

11.7.1 Construction of Weight-Balanced Trees

Binary trees require there to be a total order on the keys used to arrange the elements in the tree. Weight-balanced trees are organized by *types*, where the type is an object encapsulating the ordering relation. Creating a tree is a two-stage process. First a tree type must be created from the predicate that gives the ordering. The tree type is then used for making trees, either empty or singleton trees or trees from other aggregate structures like association lists. Once created, a tree 'knows' its type and the type is used to test compatibility between trees in operations taking two trees. Usually a small number of tree types are created at the beginning of a program and used many times throughout the program's execution.

`make-wt-tree-type` *key<?* [procedure]

This procedure creates and returns a new tree type based on the ordering predicate *key<?*. *Key<?* must be a total ordering, having the property that for all key values `a`, `b` and `c`:

```
(key<? a a)                              ⇒ #f
(and (key<? a b) (key<? b a))            ⇒ #f
(if (and (key<? a b) (key<? b c))
    (key<? a c)
    #t)                                  ⇒ #t
```

Two key values are assumed to be equal if neither is less than the other by *key<?*.

Each call to `make-wt-tree-type` returns a distinct value, and trees are only compatible if their tree types are `eq?`. A consequence is that trees that are intended to be used in binary-tree operations must all be created with a tree type originating from the same call to `make-wt-tree-type`.

`number-wt-type` [variable]

A standard tree type for trees with numeric keys. `Number-wt-type` could have been defined by

```
(define number-wt-type (make-wt-tree-type  <))
```

`string-wt-type` [variable]
A standard tree type for trees with string keys. `String-wt-type` could have been defined by

```
(define string-wt-type (make-wt-tree-type  string<?))
```

`make-wt-tree` *wt-tree-type* [procedure]
This procedure creates and returns a newly allocated weight-balanced tree. The tree is empty, i.e. it contains no associations. *Wt-tree-type* is a weight-balanced tree type obtained by calling `make-wt-tree-type`; the returned tree has this type.

`singleton-wt-tree` *wt-tree-type key datum* [procedure]
This procedure creates and returns a newly allocated weight-balanced tree. The tree contains a single association, that of *datum* with *key*. *Wt-tree-type* is a weight-balanced tree type obtained by calling `make-wt-tree-type`; the returned tree has this type.

`alist->wt-tree` *tree-type alist* [procedure]
Returns a newly allocated weight-balanced tree that contains the same associations as *alist*. This procedure is equivalent to:

```
(lambda (type alist)
  (let ((tree (make-wt-tree type)))
    (for-each (lambda (association)
                (wt-tree/add! tree
                              (car association)
                              (cdr association)))
              alist)
    tree))
```

11.7.2 Basic Operations on Weight-Balanced Trees

This section describes the basic tree operations on weight-balanced trees. These operations are the usual tree operations for insertion, deletion and lookup, some predicates and a procedure for determining the number of associations in a tree.

`wt-tree?` *object* [procedure]
Returns `#t` if *object* is a weight-balanced tree, otherwise returns `#f`.

`wt-tree/empty?` *wt-tree* [procedure]
Returns `#t` if *wt-tree* contains no associations, otherwise returns `#f`.

`wt-tree/size` *wt-tree* [procedure]
Returns the number of associations in *wt-tree*, an exact non-negative integer. This operation takes constant time.

`wt-tree/add` *wt-tree key datum* [procedure]
Returns a new tree containing all the associations in *wt-tree* and the association of *datum* with *key*. If *wt-tree* already had an association for *key*, the new association overrides the old. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

`wt-tree/add!` *wt-tree key datum* [procedure]
Associates *datum* with *key* in *wt-tree* and returns an unspecified value. If *wt-tree* already has an association for *key*, that association is replaced. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

`wt-tree/member?` *key wt-tree* [procedure]
Returns `#t` if *wt-tree* contains an association for *key*, otherwise returns `#f`. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

`wt-tree/lookup` *wt-tree key default* [procedure]
Returns the datum associated with *key* in *wt-tree*. If *wt-tree* doesn't contain an association for *key*, *default* is returned. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

`wt-tree/delete` *wt-tree key* [procedure]
Returns a new tree containing all the associations in *wt-tree*, except that if *wt-tree* contains an association for *key*, it is removed from the result. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

`wt-tree/delete!` *wt-tree key* [procedure]
If *wt-tree* contains an association for *key* the association is removed. Returns an unspecified value. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in *wt-tree*.

11.7.3 Advanced Operations on Weight-Balanced Trees

In the following the *size* of a tree is the number of associations that the tree contains, and a *smaller* tree contains fewer associations.

`wt-tree/split<` *wt-tree bound* [procedure]
Returns a new tree containing all and only the associations in *wt-tree* that have a key that is less than *bound* in the ordering relation of the tree type of *wt-tree*. The average and worst-case times required by this operation are proportional to the logarithm of the size of *wt-tree*.

`wt-tree/split>` *wt-tree bound* [procedure]
Returns a new tree containing all and only the associations in *wt-tree* that have a key that is greater than *bound* in the ordering relation of the tree type of *wt-tree*. The average and worst-case times required by this operation are proportional to the logarithm of the size of *wt-tree*.

`wt-tree/union` *wt-tree-1 wt-tree-2* [procedure]
Returns a new tree containing all the associations from both trees. This operation is asymmetric: when both trees have an association for the same key, the returned tree associates the datum from *wt-tree-2* with the key. Thus if the trees are viewed as discrete maps then `wt-tree/union` computes the map override of *wt-tree-1* by

wt-tree-2. If the trees are viewed as sets the result is the set union of the arguments. The worst-case time required by this operation is proportional to the sum of the sizes of both trees. If the minimum key of one tree is greater than the maximum key of the other tree then the worst-case time required is proportional to the logarithm of the size of the larger tree.

`wt-tree/intersection` *wt-tree-1 wt-tree-2* [procedure]
Returns a new tree containing all and only those associations from *wt-tree-1* that have keys appearing as the key of an association in *wt-tree-2*. Thus the associated data in the result are those from *wt-tree-1*. If the trees are being used as sets the result is the set intersection of the arguments. As a discrete map operation, `wt-tree/intersection` computes the domain restriction of *wt-tree-1* to (the domain of) *wt-tree-2*. The worst-case time required by this operation is proportional to the sum of the sizes of the trees.

`wt-tree/difference` *wt-tree-1 wt-tree-2* [procedure]
Returns a new tree containing all and only those associations from *wt-tree-1* that have keys that *do not* appear as the key of an association in *wt-tree-2*. If the trees are viewed as sets the result is the asymmetric set difference of the arguments. As a discrete map operation, it computes the domain restriction of *wt-tree-1* to the complement of (the domain of) *wt-tree-2*. The worst-case time required by this operation is proportional to the sum of the sizes of the trees.

`wt-tree/subset?` *wt-tree-1 wt-tree-2* [procedure]
Returns `#t` iff the key of each association in *wt-tree-1* is the key of some association in *wt-tree-2*, otherwise returns `#f`. Viewed as a set operation, `wt-tree/subset?` is the improper subset predicate. A proper subset predicate can be constructed:

```
(define (proper-subset? s1 s2)
  (and (wt-tree/subset? s1 s2)
       (< (wt-tree/size s1) (wt-tree/size s2))))
```

As a discrete map operation, `wt-tree/subset?` is the subset test on the domain(s) of the map(s). In the worst-case the time required by this operation is proportional to the size of *wt-tree-1*.

`wt-tree/set-equal?` *wt-tree-1 wt-tree-2* [procedure]
Returns `#t` iff for every association in *wt-tree-1* there is an association in *wt-tree-2* that has the same key, and *vice versa*.

Viewing the arguments as sets, `wt-tree/set-equal?` is the set equality predicate. As a map operation it determines if two maps are defined on the same domain.

This procedure is equivalent to

```
(lambda (wt-tree-1 wt-tree-2)
  (and (wt-tree/subset? wt-tree-1 wt-tree-2
       (wt-tree/subset? wt-tree-2 wt-tree-1)))
```

In the worst case the time required by this operation is proportional to the size of the smaller tree.

`wt-tree/fold` *combiner initial wt-tree* [procedure]

This procedure reduces *wt-tree* by combining all the associations, using an reverse in-order traversal, so the associations are visited in reverse order. *Combiner* is a procedure of three arguments: a key, a datum and the accumulated result so far. Provided *combiner* takes time bounded by a constant, `wt-tree/fold` takes time proportional to the size of *wt-tree*.

A sorted association list can be derived simply:

```
(wt-tree/fold (lambda (key datum list)
                (cons (cons key datum) list))
              '()
              wt-tree))
```

The data in the associations can be summed like this:

```
(wt-tree/fold (lambda (key datum sum) (+ sum datum))
              0
              wt-tree)
```

`wt-tree/for-each` *action wt-tree* [procedure]

This procedure traverses *wt-tree* in order, applying *action* to each association. The associations are processed in increasing order of their keys. *Action* is a procedure of two arguments that takes the key and datum respectively of the association. Provided *action* takes time bounded by a constant, `wt-tree/for-each` takes time proportional to the size of *wt-tree*. The example prints the tree:

```
(wt-tree/for-each (lambda (key value)
                    (display (list key value)))
                  wt-tree))
```

`wt-tree/union-merge` *wt-tree-1 wt-tree-2 merge* [procedure]

Returns a new tree containing all the associations from both trees. If both trees have an association for the same key, the datum associated with that key in the result tree is computed by applying the procedure *merge* to the key, the value from *wt-tree-1* and the value from *wt-tree-2*. *Merge* is of the form

```
(lambda (key datum-1 datum-2) ...)
```

If some key occurs only in one tree, that association will appear in the result tree without being processed by *merge*, so for this operation to make sense, either *merge* must have both a right and left identity that correspond to the association being absent in one of the trees, or some guarantee must be made, for example, all the keys in one tree are known to occur in the other.

These are all reasonable procedures for *merge*

```
(lambda (key val1 val2) (+ val1 val2))
(lambda (key val1 val2) (append val1 val2))
(lambda (key val1 val2) (wt-tree/union val1 val2))
```

However, a procedure like

```
(lambda (key val1 val2) (- val1 val2))
```

would result in a subtraction of the data for all associations with keys occuring in both trees but associations with keys occuring in only the second tree would be copied, not

negated, as is presumably be intent. The programmer might ensure that this never happens.

This procedure has the same time behavior as `wt-tree/union` but with a slightly worse constant factor. Indeed, `wt-tree/union` might have been defined like this:

```
(define (wt-tree/union tree1 tree2)
  (wt-tree/union-merge tree1 tree2
                       (lambda (key val1 val2) val2)))
```

The *merge* procedure takes the *key* as a parameter in case the data are not independent of the key.

11.7.4 Indexing Operations on Weight-Balanced Trees

Weight-balanced trees support operations that view the tree as sorted sequence of associations. Elements of the sequence can be accessed by position, and the position of an element in the sequence can be determined, both in logarthmic time.

`wt-tree/index` *wt-tree index* [procedure]
`wt-tree/index-datum` *wt-tree index* [procedure]
`wt-tree/index-pair` *wt-tree index* [procedure]

Returns the 0-based *index*th association of *wt-tree* in the sorted sequence under the tree's ordering relation on the keys. `wt-tree/index` returns the *index*th key, `wt-tree/index-datum` returns the datum associated with the *index*th key and `wt-tree/index-pair` returns a new pair `(`*key* `.` *datum*`)` which is the `cons` of the *index*th key and its datum. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree.

These operations signal a condition of type `condition-type:bad-range-argument` if *index*`<0` or if *index* is greater than or equal to the number of associations in the tree. If the tree is empty, they signal an anonymous error.

Indexing can be used to find the median and maximum keys in the tree as follows:

```
median:   (wt-tree/index wt-tree
                         (quotient (wt-tree/size wt-tree)
                                   2))
maximum:  (wt-tree/index wt-tree
                         (- (wt-tree/size wt-tree)
                            1))
```

`wt-tree/rank` *wt-tree key* [procedure]

Determines the 0-based position of *key* in the sorted sequence of the keys under the tree's ordering relation, or `#f` if the tree has no association with for *key*. This procedure returns either an exact non-negative integer or `#f`. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree.

`wt-tree/min` *wt-tree* [procedure]
`wt-tree/min-datum` *wt-tree* [procedure]

`wt-tree/min-pair` *wt-tree* [procedure]
Returns the association of *wt-tree* that has the least key under the tree's ordering relation. `wt-tree/min` returns the least key, `wt-tree/min-datum` returns the datum associated with the least key and `wt-tree/min-pair` returns a new pair `(key . datum)` which is the `cons` of the minimum key and its datum. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree.

These operations signal an error if the tree is empty. They could have been written

```
(define (wt-tree/min tree)
  (wt-tree/index tree 0))
(define (wt-tree/min-datum tree)
  (wt-tree/index-datum tree 0))
(define (wt-tree/min-pair tree)
  (wt-tree/index-pair tree 0))
```

`wt-tree/delete-min` *wt-tree* [procedure]
Returns a new tree containing all of the associations in *wt-tree* except the association with the least key under the *wt-tree*'s ordering relation. An error is signalled if the tree is empty. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree. This operation is equivalent to

```
(wt-tree/delete wt-tree (wt-tree/min wt-tree))
```

`wt-tree/delete-min!` *wt-tree* [procedure]
Removes the association with the least key under the *wt-tree*'s ordering relation. An error is signalled if the tree is empty. The average and worst-case times required by this operation are proportional to the logarithm of the number of associations in the tree. This operation is equivalent to

```
(wt-tree/delete! wt-tree (wt-tree/min wt-tree))
```

11.8 Associative Maps

Starting with version 11.1, MIT/GNU Scheme provides an abstract associative-map interface that can be backed by any kind of association mechanism. The interface is similar to that of SRFI 125. Associative maps can be mutable or immutable, depending on the backing implementation. As of this writing we support the following implementations: alists, weak alists, red/black trees, hash tables, and tries, all of which are mutable.

11.8.1 Amap constructors

All associative-map constructors take a comparator argument followed by a list of additional specification arguments. In all cases, the comparator is used to compare keys in the map, while the additional arguments specify the implementation and/or features of the map being created.

`make-amap` *comparator arg . . .* [procedure]
Creates a new mutable associative map as specified by *comparator* and *arg*s.

`alist->amap` *alist comparator arg ...* [procedure]
Creates a new associative map as specified by *comparator* and *arg*s, and which contains the associations in *alist*.

`amap-unfold` *stop? mapper successor seed comparator arg ...* [procedure]
Creates a new associative map as specified by *comparator* and *arg*s. The associations in the resulting map are generated using the additional arguments.

The *stop?* argument is a unary predicate that takes a state value and returns `#t` if generation is complete, otherwise `#f`. The *mapper* argument is a unary procedure that takes a state value and returns two values: a key and a value. The *successor* argument is a unary procedure that takes a state value and returns a new state value. And *seed* is the initial state value.

The process by which the generator adds associations to the map is this:

```
(let loop ((state seed))
  (if (stop? state)
      result
      (let-values (((key value) (mapper state)))
        (amap-set! result key value)
        (loop (successor state)))))
```

The arguments passed to a constructor can be divided into categories:

- Each backing implementation has an associated name, which can be passed as one of the arguments. At present this includes `alist`, `hash-table`, `red/black-tree`, etc.

  There must be at most one such name; it is an error to pass two or more implementation names. If an implementation name is given, then the remaining arguments must be supported by that implementation.
- The arguments `weak-keys`, `weak-values`, `ephemeral-keys`, and `ephemeral-values` specify how associations interact with garbage collection. Not all implementations support these arguments, and those that do may support only some of them. Some combinations are not allowed: for example, `weak-keys` and `ephemeral-keys` are mutually exclusive.
- Additional arguments are used to guide the choice of implementation when it is not explicitly specified. The desired time complexity can be specified: `linear-time`, `log-time`, `amortized-constant-time`, and `sublinear-time`. The argument `ordered-by-key` specifies that the map keeps its associations ordered.
- The argument `thread-safe` specifies that the implementation is safe to use in a multi-threaded environment. At present none of the implementations are thread-safe.
- Finally, an exact non-negative integer argument specifies the initial size of the map.

This is a complex set of possible arguments. In order to help explore what arguments can be used, and in what combinations, we provide some utility procedures:

`amap-implementation-names` [procedure]
Returns a list of the supported implementation names.

`amap-implementation-supported-args` *name* [procedure]
Returns a list of the arguments supported by the implementation specified by *name*. This list can include the procedure `exact-nonnegative-integer?` if the implementation supports an initial size.

`amap-implementation-supports-args?` *name args* [procedure]
Returns `#t` if the implementation specified by *name* supports *args*, otherwise returns `#f`.

An implementation may support a limited set of comparators. For example, a hash table requires a comparator that satisfies `comparator-hashable?`, while a binary tree requires one satisfying `comparator-ordered?`.

`amap-implementation-supports-comparator?` *name comparator* [procedure]
Returns `#t` if the implementation specified by *name* supports *comparator*, otherwise returns `#f`.

11.8.2 Amap predicates

`amap?` *object* [procedure]
True iff *object* is an associative map.

`amap-contains?` *amap key* [procedure]
True iff *amap* contains an association for *key*.

`amap-empty?` *amap* [procedure]
True iff *amap* has no associations.

`amap=?` *value-comparator amap1 amap2* [procedure]
True iff the associations in *amap1* are the same as the associations in *amap2*. This means that *amap1* and *amap2* have the same keys (in the sense of their shared equality predicate), and that for each key they have the same value (in the sense of *value-comparator*).

`amap-mutable?` *amap* [procedure]
True iff *amap* is mutable.

11.8.3 Amap accessors

`amap-ref` *amap key* [*fail* [*succeed*]] [procedure]
Looks up the value associated to *key* in *amap*, invokes the procedure *succeed* on it, and returns its result; if *succeed* is not provided, then the value itself is returned. If *key* is not contained in *amap* and *fail* is supplied, then *fail* is invoked on no arguments and its result is returned. Otherwise an error is signaled.

`amap-ref/default` *amap key default* [procedure]
Semantically equivalent to, but may be more efficient than, the following code:

```
(amap-ref amap key (lambda () default))
```

`amap-comparator` *amap* [procedure]
Returns the comparator that was used to create *amap*.

`amap-args` *amap* [procedure]
Returns the extra arguments that were used to create *amap*.

`amap-implementation-name` *amap* [procedure]
Returns the name of *amap*'s backing implementation.

11.8.4 Amap mutators

`amap-set!` *amap* [*key value*] ... [procedure]
Repeatedly mutates *amap*, creating new associations in it by processing the arguments from left to right. The args alternate between keys and values. Whenever there is a previous association for a key, it is deleted. It is an error if a key does not satisfy the type check procedure of the comparator of *amap*. Likewise, it is an error if a key is not a valid argument to the equality predicate of *amap*. Returns an unspecified value.

`amap-delete!` *amap key* ... [procedure]
Deletes any association to each *key* in *amap* and returns the number of keys that had associations.

`amap-intern!` *amap key fail* [procedure]
Effectively invokes `amap-ref` with the given arguments and returns what it returns. If *key* was not found in *amap*, its value is set to the result of calling *fail*.

`amap-update!` *amap key updater* [*fail* [*succeed*]] [procedure]
Semantically equivalent to, but may be more efficient than, the following code:

```
(amap-set! amap key (updater (amap-ref amap key fail succeed)))
```

`amap-update!/default` *amap key updater default* [procedure]
Semantically equivalent to, but may be more efficient than, the following code:

```
(amap-set! amap key (updater (amap-ref/default amap key default)))
```

`amap-pop!` *amap* [procedure]
Chooses an arbitrary association from *amap* and removes it, returning the key and value as two values. Signals an error if *amap* is empty.

`amap-clear!` *amap* [procedure]
Delete all the associations from *amap*.

`amap-clean!` *amap* [procedure]
If *amap* has weak or ephemeral associations, cleans up any storage for associations whose key and/or value has been reclaimed by the garbage collector. Otherwise does nothing.

This procedure does not have any visible effect, since the associations it cleans up are ignored by all other parts of the interface. Additionally, most implementations clean up their storage incrementally as they are used. But this procedure provides for edge cases where the reclaimed storage might matter.

11.8.5 Amap mapping and folding

`amap-map` *procedure amap* [*comparator arg ...*] [procedure]
Returns a newly allocated associative map as if by `(make-amap` *`comparator arg ...`*`)`. Calls *procedure* for every association in *amap* with the value of the association. The key of the association and the result of invoking *procedure* are entered into the new map.

If *comparator* recognizes multiple keys in *amap* as equivalent, any one of such associations is taken.

`amap-for-each` *procedure amap* [procedure]
Calls *procedure* for every association in *amap* with two arguments: the key of the association and the value of the association. The value returned by *procedure* is discarded. Returns an unspecified value.

`amap-map!` *procedure amap* [procedure]
Calls *procedure* for every association in *amap* with two arguments: the key of the association and the value of the association. The value returned by *procedure* is used to update the value of the association. Returns an unspecified value.

`amap-map->list` *procedure amap* [procedure]
Calls *procedure* for every association in *amap* with two arguments: the key of the association and the value of the association. The values returned by the invocations of *procedure* are accumulated into a list, which is returned.

`amap-fold` *kons knil amap* [procedure]
Calls *kons* for every association in *amap* with three arguments: the key of the association, the value of the association, and an accumulated value *val*. The argument *knil* is seed for the first invocation of *kons*, and for subsequent invocations of *kons*, the returned value of the previous invocation. The value returned by `amap-fold` is the return value of the last invocation of *kons*.

`amap-prune!` *predicate amap* [procedure]
Calls *predicate* for every association in *amap* with two arguments, the key and the value of the association, and removes all associations from *amap* for which *predicate* returns true. Returns an unspecified value.

11.8.6 Amap contents

`amap-size` *amap* [procedure]
Returns the number of associations in *amap* as an exact integer.

`amap-keys` *amap* [procedure]
Returns a newly allocated list of all the keys in *amap*.

`amap-values` *amap* [procedure]
Returns a newly allocated list of all the values in *amap*.

`amap-entries` *amap* [procedure]
Returns two values, a newly allocated list of all the keys in *amap* and a newly allocated list of all the values in *amap* in the corresponding order.

`amap-find` *procedure amap fail* [procedure]
For each association of *amap*, invoke *procedure* on its key and value. If *procedure* returns true, then `amap-find` returns what *procedure* returns. If all the calls to *procedure* return `#f`, returns the result of invoking the thunk *fail*.

`amap-count` *predicate amap* [procedure]
For each association of *amap*, invoke *predicate* on its key and value. Returns the number of calls to *predicate* that returned true.

11.8.7 Amap copying and conversion

`amap-copy` *amap* [*mutable?*] [procedure]
Returns a newly allocated associative map with the same properties and associations as *amap*. If *mutable?* is given and is true, the new associative map is mutable. Otherwise it is immutable provided that the implementation supports immutable maps.

`amap-empty-copy` *amap* [procedure]
Returns a newly allocated mutable associative map with the same properties as *amap*, but with no associations.

`amap->alist` *amap* [procedure]
Returns an alist with the same associations as *amap* in an unspecified order.

11.8.8 Amaps as sets

`amap-union!` *amap1 amap2* [procedure]
Adds the associations of *amap2* to *amap1* and returns *amap1*. If a key appears in both maps, its value is set to the value appearing in *amap1*.

`amap-intersection!` *amap1 amap2* [procedure]
Deletes the associations from *amap1* whose keys don't also appear in *amap2* and returns *amap1*.

`amap-difference!` *amap1 amap2* [procedure]
Deletes the associations of *amap1* whose keys are also present in *amap2* and returns *amap1*.

`amap-xor!` *amap1 amap2* [procedure]
Deletes the associations of *amap1* whose keys are also present in *amap2*, and then adds the associations of *amap2* whose keys are not present in *amap1* to *amap1*. Returns *amap1*.

12 Procedures

Procedures are created by evaluating `lambda` expressions (see Section 2.1 [Lambda Expressions], page 15); the `lambda` may either be explicit or may be implicit as in a "procedure `define`" (see Section 2.4 [Definitions], page 22). Also there are special built-in procedures, called *primitive procedures*, such as `car`; these procedures are not written in Scheme but in the language used to implement the Scheme system. MIT/GNU Scheme also provides *application hooks*, which support the construction of data structures that act like procedures.

In MIT/GNU Scheme, the written representation of a procedure tells you the type of the procedure (compiled, interpreted, or primitive):

```
pp
     ⇒  #[compiled-procedure 56 ("pp" #x2) #x10 #x307578]
(lambda (x) x)
     ⇒  #[compound-procedure 57]
(define (foo x) x)
foo
     ⇒  #[compound-procedure 58 foo]
car
     ⇒  #[primitive-procedure car]
(call-with-current-continuation (lambda (x) x))
     ⇒  #[continuation 59]
```

Note that interpreted procedures are called "compound" procedures (strictly speaking, compiled procedures are also compound procedures). The written representation makes this distinction for historical reasons, and may eventually change.

12.1 Procedure Operations

`apply` *procedure object object . . .* [procedure]

Calls *procedure* with the elements of the following list as arguments:

```
(cons* object object ...)
```

The initial *object*s may be any objects, but the last *object* (there must be at least one *object*) must be a list.

```
(apply + (list 3 4 5 6))                ⇒  18
(apply + 3 4 '(5 6))                    ⇒  18

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))
((compose sqrt *) 12 75)                ⇒  30
```

`procedure?` *object* [procedure]

Returns `#t` if *object* is a procedure; otherwise returns `#f`. If `#t` is returned, exactly one of the following predicates is satisfied by *object*: `compiled-procedure?`, `compound-procedure?`, or `primitive-procedure?`.

`compiled-procedure?` *object* [procedure]
Returns `#t` if *object* is a compiled procedure; otherwise returns `#f`.

`compound-procedure?` *object* [procedure]
Returns `#t` if *object* is a compound (i.e. interpreted) procedure; otherwise returns `#f`.

`primitive-procedure?` *object* [procedure]
Returns `#t` if *object* is a primitive procedure; otherwise returns `#f`.

`procedure-environment` *procedure* [procedure]
Returns the closing environment of *procedure*. Signals an error if *procedure* is a primitive procedure, or if *procedure* is a compiled procedure for which the debugging information is unavailable.

12.2 Arity

Each procedure has an *arity*, which is the minimum and (optionally) maximum number of arguments that it will accept. MIT/GNU Scheme provides an abstraction that represents arity, and tests for the apparent arity of a procedure.

Arity objects come in two forms: the simple form, an exact non-negative integer, represents a fixed number of arguments. The general form is a pair whose `car` represents the minimum number of arguments and whose `cdr` is the maximum number of arguments.

`make-procedure-arity` *min* [*max* [*simple-ok?*]] [procedure]
Returns an arity object made from *min* and *max*. *Min* must be an exact non-negative integer. *Max* must be an exact non-negative integer at least as large as *min*. Alternatively, *max* may be omitted or given as '`#f`', which represents an arity with no upper bound.

If *simple-ok?* is true, the returned arity is in the simple form (an exact non-negative integer) when possible, and otherwise is always in the general form. *Simple-ok?* defaults to '`#f`'.

`procedure-arity?` *object* [procedure]
Returns '`#t`' if *object* is an arity object, and '`#f`' otherwise.

`procedure-arity-min` *arity* [procedure]
`procedure-arity-max` *arity* [procedure]
Return the lower and upper bounds of *arity*, respectively.

The following procedures test for the apparent arity of a procedure. The results of the test may be less restrictive than the effect of calling the procedure. In other words, these procedures may indicate that the procedure will accept a given number of arguments, but if you call the procedure it may signal a `condition-type:wrong-number-of-arguments` error. For example, here is a procedure that appears to accept any number of arguments, but when called will signal an error if the number of arguments is not one:

```
(lambda arguments (apply car arguments))
```

`procedure-arity` *procedure* [procedure]
Returns the arity that *procedure* accepts. The result may be in either simple or general form.

```
(procedure-arity (lambda () 3))         ⇒  (0 . 0)
(procedure-arity (lambda (x) x))        ⇒  (1 . 1)
(procedure-arity car)                   ⇒  (1 . 1)
(procedure-arity (lambda x x))          ⇒  (0 . #f)
(procedure-arity (lambda (x . y) x))    ⇒  (1 . #f)
(procedure-arity (lambda (x #!optional y) x))
                                        ⇒  (1 . 2)
```

`procedure-arity-valid?` *procedure arity* [procedure]
Returns '`#t`' if *procedure* accepts *arity*, and '`#f`' otherwise.

`procedure-of-arity?` *object arity* [procedure]
Returns '`#t`' if *object* is a procedure that accepts *arity*, and '`#f`' otherwise. Equivalent to:

```
(and (procedure? object)
     (procedure-arity-valid? object arity))
```

`guarantee-procedure-of-arity` *object arity caller* [procedure]
Signals an error if *object* is not a procedure accepting *arity*. *Caller* is a symbol that is printed as part of the error message and is intended to be the name of the procedure where the error occurs.

`thunk?` *object* [procedure]
Returns '`#t`' if *object* is a procedure that accepts zero arguments, and '`#f`' otherwise. Equivalent to:

```
(procedure-of-arity? object 0)
```

12.3 Primitive Procedures

`make-primitive-procedure` *name* [*arity*] [procedure]
Name must be a symbol. *Arity* must be an exact non-negative integer, `-1`, `#f`, or `#t`; if not supplied it defaults to `#f`. Returns the primitive procedure called *name*. May perform further actions depending on *arity*:

`#f` If the primitive procedure is not implemented, signals an error.

`#t` If the primitive procedure is not implemented, returns `#f`.

integer If the primitive procedure is implemented, signals an error if its arity is not equal to *arity*. If the primitive procedure is not implemented, returns an unimplemented primitive procedure object that accepts *arity* arguments. An *arity* of `-1` means it accepts any number of arguments.

`primitive-procedure-name` *primitive-procedure* [procedure]
Returns the name of *primitive-procedure*, a symbol.

```
(primitive-procedure-name car)          ⇒  car
```

`implemented-primitive-procedure?` *primitive-procedure* [procedure]
Returns `#t` if *primitive-procedure* is implemented; otherwise returns `#f`. Useful because the code that implements a particular primitive procedure is not necessarily linked into the executable Scheme program.

12.4 Continuations

`call-with-current-continuation` *procedure* [procedure]
Procedure must be a procedure of one argument. Packages up the current continuation (see below) as an *escape procedure* and passes it as an argument to *procedure*. The escape procedure is a Scheme procedure of one argument that, if it is later passed a value, will ignore whatever continuation is in effect at that later time and will give the value instead to the continuation that was in effect when the escape procedure was created. The escape procedure created by `call-with-current-continuation` has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

The following examples show only the most common uses of this procedure. If all real programs were as simple as these examples, there would be no need for a procedure with the power of `call-with-current-continuation`.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                                ⇒  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj) (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))
(list-length '(1 2 3 4))                ⇒  4
(list-length '(a b . c))                ⇒  #f
```

A common use of `call-with-current-continuation` is for structured, non-local exits from loops or procedure bodies, but in fact `call-with-current-continuation` is quite useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated a continuation exists that wants the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, the continuation will take the result, print it on the screen, prompt for the next input, evaluate it, and

so on forever. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top-level continuation to be printed. Normally these ubiquitous continuations are hidden behind the scenes and programmers don't think much about them. On the rare occasions that you may need to deal explicitly with continuations, `call-with-current-continuation` lets you do so by creating a procedure that acts just like the current continuation.

`continuation?` *object* [procedure]
Returns `#t` if *object* is a continuation; otherwise returns `#f`.

`within-continuation` *continuation thunk* [procedure]
Thunk must be a procedure of no arguments. Conceptually, `within-continuation` invokes *continuation* on the result of invoking *thunk*, but *thunk* is executed in the dynamic state of *continuation*. In other words, the "current" continuation is abandoned before *thunk* is invoked.

`dynamic-wind` *before thunk after* [procedure]
Calls *thunk* without arguments, returning the result(s) of this call. *Before* and *after* are called, also without arguments, as required by the following rules. Note that in the absence of calls to continuations captured using `call-with-current-continuation` the three arguments are called once each, in order. *Before* is called whenever execution enters the dynamic extent of the call to *thunk* and *after* is called whenever it exits that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. In Scheme, because of `call-with-current-continuation`, the dynamic extent of a call may not be a single, connected time period. It is defined as follows:

- The dynamic extent is entered when execution of the body of the called procedure begins.
- The dynamic extent is also entered when execution is not within the dynamic extent and a continuation is invoked that was captured (using `call-with-current-continuation`) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *after*s from these two invocations of `dynamic-wind` are both to be called, then the *after* associated with the second (inner) call to `dynamic-wind` is called first.

If a second call to `dynamic-wind` occurs within the dynamic extent of the call to *thunk* and then a continuation is invoked in such a way that the *before*s from these two invocations of `dynamic-wind` are both to be called, then the *before* associated with the first (outer) call to `dynamic-wind` is called first.

If invoking a continuation requires calling the *before* from one call to `dynamic-wind` and the *after* from another, then the *after* is called first.

The effect of using a captured continuation to enter or exit the dynamic extent of a call to *before* or *after* is undefined.

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

⇒ (connect talk1 disconnect connect talk2 disconnect)
```

The following two procedures support multiple values.

`call-with-values` *thunk procedure* [procedure]
: *Thunk* must be a procedure of no arguments, and *procedure* must be a procedure. *Thunk* is invoked with a continuation that expects to receive multiple values; specifically, the continuation expects to receive the same number of values that *procedure* accepts as arguments. *Thunk* must return multiple values using the `values` procedure. Then *procedure* is called with the multiple values as its arguments. The result yielded by *procedure* is returned as the result of `call-with-values`.

`values` *object* ... [procedure]
: Returns multiple values. The continuation in effect when this procedure is called must be a multiple-value continuation that was created by `call-with-values`. Furthermore it must accept as many values as there are *object*s.

12.5 Application Hooks

Application hooks are objects that can be applied like procedures. Each application hook has two parts: a *procedure* that specifies what to do when the application hook is applied, and an arbitrary object, called *extra*. Often the procedure uses the extra object to determine what to do.

There are two kinds of application hooks, which differ in what arguments are passed to the procedure. When an *apply hook* is applied, the procedure is passed exactly the same arguments that were passed to the apply hook. When an *entity* is applied, the entity itself is passed as the first argument, followed by the other arguments that were passed to the entity.

Both apply hooks and entities satisfy the predicate `procedure?`. Each satisfies either `compiled-procedure?`, `compound-procedure?`, or `primitive-procedure?`, depending on

its procedure component. An apply hook is considered to accept the same number of arguments as its procedure, while an entity is considered to accept one less argument than its procedure.

`make-apply-hook` *procedure object* [procedure]
Returns a newly allocated apply hook with a procedure component of *procedure* and an extra component of *object*.

`apply-hook?` *object* [procedure]
Returns `#t` if *object* is an apply hook; otherwise returns `#f`.

`apply-hook-procedure` *apply-hook* [procedure]
Returns the procedure component of *apply-hook*.

`set-apply-hook-procedure!` *apply-hook procedure* [procedure]
Changes the procedure component of *apply-hook* to be *procedure*. Returns an unspecified value.

`apply-hook-extra` *apply-hook* [procedure]
Returns the extra component of *apply-hook*.

`set-apply-hook-extra!` *apply-hook object* [procedure]
Changes the extra component of *apply-hook* to be *object*. Returns an unspecified value.

`make-entity` *procedure object* [procedure]
Returns a newly allocated entity with a procedure component of *procedure* and an extra component of *object*.

`entity?` *object* [procedure]
Returns `#t` if *object* is an entity; otherwise returns `#f`.

`entity-procedure` *entity* [procedure]
Returns the procedure component of *entity*.

`set-entity-procedure!` *entity procedure* [procedure]
Changes the procedure component of *entity* to be *procedure*. Returns an unspecified value.

`entity-extra` *entity* [procedure]
Returns the extra component of *entity*.

`set-entity-extra!` *entity object* [procedure]
Changes the extra component of *entity* to be *object*. Returns an unspecified value.

13 Environments

13.1 Environment Operations

Environments are first-class objects in MIT/GNU Scheme. An environment consists of some bindings and possibly a parent environment, from which other bindings are inherited. The operations in this section reveal the frame-like structure of environments by permitting you to examine the bindings of a particular environment separately from those of its parent.

There are several types of bindings that can occur in an environment. The most common is the simple variable binding, which associates a value (any Scheme object) with an identifier (a symbol). A variable binding can also be *unassigned*, which means that it has no value. An unassigned variable is bound, in that is will shadow other bindings of the same name in ancestor environments, but a reference to that variable will signal an error of type `condition-type:unassigned-variable`. An unassigned variable can be *assigned* (using `set!` or `environment-assign!`) to give it a value.

In addition to variable bindings, an environment can also have *keyword bindings*. A keyword binding associates a syntactic keyword (usually a macro transformer) with an identifier. Keyword bindings are special in that they are considered "bound", but ordinary variable references don't work on them. So an attempt to reference or assign a keyword binding results in an error of type `condition-type:macro-binding`. However, keyword bindings can be redefined using `define` or `environment-define`.

`environment?` *object* [procedure]
Returns `#t` if *object* is an environment; otherwise returns `#f`.

`environment-has-parent?` *environment* [procedure]
Returns `#t` if *environment* has a parent environment; otherwise returns `#f`.

`environment-parent` *environment* [procedure]
Returns the parent environment of *environment*. It is an error if *environment* has no parent.

`environment-bound-names` *environment* [procedure]
Returns a newly allocated list of the names (symbols) that are bound by *environment*. This does not include the names that are bound by the parent environment of *environment*. It does include names that are unassigned or keywords in *environment*.

`environment-macro-names` *environment* [procedure]
Returns a newly allocated list of the names (symbols) that are bound to syntactic keywords in *environment*.

`environment-bindings` *environment* [procedure]
Returns a newly allocated list of the bindings of *environment*; does not include the bindings of the parent environment. Each element of this list takes one of two forms: `(`*`symbol`*`)` indicates that *symbol* is bound but unassigned, while `(`*`symbol object`*`)` indicates that *symbol* is bound, and its value is *object*.

`environment-reference-type` *environment symbol* [procedure]
Returns a symbol describing the *reference type* of *symbol* in *environment* or one of its ancestor environments. The result is one of the following:

`normal` means *symbol* is a variable binding with a normal value.

`unassigned`
means *symbol* is a variable binding with no value.

`macro` means *symbol* is a keyword binding.

`unbound` means *symbol* has no associated binding.

`environment-bound?` *environment symbol* [procedure]
Returns `#t` if *symbol* is bound in *environment* or one of its ancestor environments; otherwise returns `#f`. This is equivalent to

```
(not (eq? 'unbound
          (environment-reference-type environment symbol)))
```

`environment-assigned?` *environment symbol* [procedure]
Returns `#t` if *symbol* is bound in *environment* or one of its ancestor environments, and has a normal value. Returns `#f` if it is bound but unassigned. Signals an error if it is unbound or is bound to a keyword.

`environment-lookup` *environment symbol* [procedure]
Symbol must be bound to a normal value in *environment* or one of its ancestor environments. Returns the value to which it is bound. Signals an error if unbound, unassigned, or a keyword.

`environment-lookup-macro` *environment symbol* [procedure]
If *symbol* is a keyword binding in *environment* or one of its ancestor environments, returns the value of the binding. Otherwise, returns `#f`. Does not signal any errors other than argument-type errors.

`environment-assignable?` *environment symbol* [procedure]
Symbol must be bound in *environment* or one of its ancestor environments. Returns `#t` if the binding may be modified by side effect.

`environment-assign!` *environment symbol object* [procedure]
Symbol must be bound in *environment* or one of its ancestor environments, and must be assignable. Modifies the binding to have *object* as its value, and returns an unspecified result.

`environment-definable?` *environment symbol* [procedure]
Returns `#t` if *symbol* is definable in *environment*, and `#f` otherwise. At present, this is false for environments generated by application of compiled procedures, and true for all other environments.

`environment-define` *environment symbol object* [procedure]
Defines *symbol* to be bound to *object* in *environment*, and returns an unspecified value. Signals an error if *symbol* isn't definable in *environment*.

`environment-define-macro` *environment symbol transformer* [procedure]
Defines *symbol* to be a keyword bound to *transformer* in *environment*, and returns an unspecified value. Signals an error if *symbol* isn't definable in *environment*. The type of *transformer* is defined by the syntax engine and is not checked by this procedure. If the type is incorrect this will subsequently signal an error during syntax expansion.

`eval` *expression environment* [procedure]
Evaluates *expression*, a list-structure representation (sometimes called s-expression representation) of a Scheme expression, in *environment*. You rarely need `eval` in ordinary programs; it is useful mostly for evaluating expressions that have been created "on the fly" by a program. `eval` is relatively expensive because it must convert *expression* to an internal form before it is executed.

```
(define foo (list '+ 1 2))
(eval foo (the-environment))            ⇒  3
```

13.2 Environment Variables

The `user-initial-environment` is where the top-level read-eval-print (REP) loop evaluates expressions and binds definitions. It is a child of `system-global-environment`, which is where all of the Scheme system definitions are bound. All of the bindings in `system-global-environment` are available when the current environment is `user-initial-environment`. However, any new bindings that you create in the REP loop (with `define` forms or by loading files containing `define` forms) occur in `user-initial-environment`.

`system-global-environment` [variable]
The variable `system-global-environment` is bound to the distinguished environment that's the ancestor of most other environments (except for those created by `make-root-top-level-environment`). It is the parent environment of `user-initial-environment`. Primitives, system procedures, and most syntactic keywords are bound (and sometimes closed) in this environment.

`user-initial-environment` [variable]
The variable `user-initial-environment` is bound to the default environment in which typed expressions are evaluated by the top-level REP loop.

Although all bindings in `system-global-environment` are visible to the REP loop, definitions that are typed at, or loaded by, the REP loop occur in the `user-initial-environment`. This is partly a safety measure: if you enter a definition that happens to have the same name as a critical system procedure, your definition will be visible only to the procedures you define in the `user-initial-environment`; the MIT/GNU Scheme system procedures, which are defined in `system-global-environment`, will continue to see the original definition.

13.3 REPL Environment

`nearest-repl/environment` [procedure]
Returns the current REP loop environment (i.e. the current environment of the closest enclosing REP loop). When Scheme first starts up, this is the same as `user-initial-environment`.

`ge` *environment* [procedure]
Changes the current REP loop environment to *environment*. *Environment* can be either an environment or a procedure object. If it's a procedure, the environment in which that procedure was closed is the new environment.

13.4 Top-level Environments

The operations in this section manipulate *top-level environments*, as opposed to environments created by the application of procedures. For historical reasons, top-level environments are referred to as *interpreter environments*.

`the-environment` [special form]
Returns the current environment. This form may only be evaluated in a top-level environment. An error is signalled if it appears elsewhere.

`top-level-environment?` *object* [procedure]
`interpreter-environment?` *object* [procedure]
Returns `#t` if *object* is an top-level environment; otherwise returns `#f`.
`interpreter-environment?` is an alias for `top-level-environment?`.

`extend-top-level-environment` *environment* [*names* [*values*]] [procedure]
`make-top-level-environment` [*names* [*values*]] [procedure]
`make-root-top-level-environment` [*names* [*values*]] [procedure]
Returns a newly allocated top-level environment. `extend-top-level-environment` creates an environment that has parent *environment*, `make-top-level-environment` creates an environment that has parent `system-global-environment`, and `make-root-top-level-environment` creates an environment that has no parent.
The optional arguments *names* and *values* are used to specify initial bindings in the new environment. If specified, *names* must be a list of symbols, and *values* must be a list of objects. If only *names* is specified, each name in *names* will be bound in the environment, but unassigned. If *names* and *values* are both specified, they must be the same length, and each name in *names* will be bound to the corresponding value in *values*. If neither *names* nor *values* is specified, the environment will have no initial bindings.

`link-variables` *environment1 symbol1 environment2 symbol2* [procedure]
Defines *symbol1* in *environment1* to have the same binding as *symbol2* in *environment2*, and returns an unspecified value. Prior to the call, *symbol2* must be bound in *environment2*, but the type of binding is irrelevant; it may be a normal binding, an unassigned binding, or a keyword binding. Signals an error if *symbol1* isn't definable in *environment1*, or if *symbol2* is unbound in *environment2*.
By "the same binding", we mean that the value cell is shared between the two environments. If a value is assigned to *symbol1* in *environment1*, a subsequent reference to *symbol2* in *environment2* will see that value, and vice versa.

`unbind-variable` *environment symbol* [procedure]
If *symbol* is bound in *environment* or one of its ancestor environments, removes the binding, so that subsequent accesses to that symbol behave as if the binding never existed. Returns `#t` if there was a binding prior to the call, and `#f` if there wasn't.

14 Input/Output

This chapter describes the procedures that are used for input and output (I/O). The chapter first describes *ports* and how they are manipulated, then describes the I/O operations. Finally, some low-level procedures are described that permit the implementation of custom ports and high-performance I/O.

14.1 Ports

Ports represent input and output devices. To Scheme, an *input port* is a Scheme object that can deliver data upon command, while an *output port* is a Scheme object that can accept data. Whether the input and output port types are disjoint is implementation-dependent. (In MIT/GNU Scheme, there are input ports, output ports, and input/output ports.)

Different port types operate on different data. Scheme implementations are required to support textual ports and binary ports, but may also provide other port types.

A *textual port* supports reading or writing of individual characters from or to a backing store containing characters using `read-char` and `write-char` below, and it supports operations defined in terms of characters, such as `read` and `write`.

A *binary port* supports reading or writing of individual bytes from or to a backing store containing bytes using `read-u8` and `write-u8` below, as well as operations defined in terms of bytes. Whether the textual and binary port types are disjoint is implementation-dependent. (In MIT/GNU Scheme, textual ports and binary ports are distinct.)

Ports can be used to access files, devices, and similar things on the host system on which the Scheme program is running.

`call-with-port` *port procedure* [standard procedure]

It is an error if *procedure* does not accept one argument.

The `call-with-port` procedure calls *procedure* with *port* as an argument. If *procedure* returns, then the port is closed automatically and the values yielded by *procedure* are returned. If *procedure* does not return, then the port must not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

Rationale: Because Scheme's escape procedures have unlimited extent, it is possible to escape from the current continuation but later to resume it. If implementations were permitted to close the port on any escape from the current continuation, then it would be impossible to write portable code using both `call-with-current-continuation` and `call-with-port`.

`call-with-truncated-output-port` *limit output-port procedure* [procedure]

The *limit* argument must be a nonnegative integer. It is an error if *procedure* does not accept one argument.

This procedure uses a continuation to escape from *procedure* if it tries to write more than *limit* characters.

It calls *procedure* with a special output port as an argument. Up to *limit* characters may be written to that output port, and those characters are transparently written through to *output-port*.

If the number of characters written to that port exceeds *limit*, then the escape continuation is invoked and `#t` is returned. Otherwise, *procedure* returns normally and `#f` is returned.

Note that if *procedure* writes exactly *limit* characters, then the escape continuation is *not* invoked, and `#f` is returned.

In no case does `call-with-truncated-output-port` close *output-port*.

`input-port?` *object* [standard procedure]
`output-port?` *object* [standard procedure]
`i/o-port?` *object* [procedure]
`textual-port?` *object* [standard procedure]
`binary-port?` *object* [standard procedure]
`port?` *object* [standard procedure]

These procedures return `#t` if *object* is an input port, output port, input/output port, textual port, binary port, or any kind of port, respectively. Otherwise they return `#f`.

`input-port-open?` *port* [standard procedure]
`output-port-open?` *port* [standard procedure]

Returns `#t` if *port* is still open and capable of performing input or output, respectively, and `#f` otherwise.

`current-input-port` [*input-port*] [standard parameter]
`current-output-port` [*output-port*] [standard parameter]
`current-error-port` [*output-port*] [standard parameter]

Returns the current default input port, output port, or error port (an output port), respectively. These procedures are parameter objects, which can be overridden with `parameterize`. The initial bindings for these are implementation-defined textual ports.

`notification-output-port` [*output-port*] [parameter]

Returns an output port suitable for generating "notifications", that is, messages to the user that supply interesting information about the execution of a program. For example, the `load` procedure writes messages to this port informing the user that a file is being loaded.

This procedure is a parameter object, which can be overridden with `parameterize`.

`trace-output-port` [*output-port*] [parameter]

Returns an output port suitable for generating "tracing" information about a program's execution. The output generated by the `trace` procedure is sent to this port.

This procedure is a parameter object, which can be overridden with `parameterize`.

`interaction-i/o-port` [*i/o-port*] [parameter]

Returns an I/O port suitable for querying or prompting the user. The standard prompting procedures use this port by default (see Section 14.11 [Prompting], page 247).

This procedure is a parameter object, which can be overridden with `parameterize`.

`close-port` *port* [standard procedure]
`close-input-port` *port* [standard procedure]
`close-output-port` *port* [standard procedure]
Closes the resource associated with *port*, rendering the port incapable of delivering or accepting data. It is an error to apply the last two procedures to a port which is not an input or output port, respectively. Scheme implementations may provide ports which are simultaneously input and output ports, such as sockets; the close-input-port and close-output-port procedures can then be used to close the input and output sides of the port independently.

These routines have no effect if the port has already been closed.

`set-current-input-port!` *input-port* [obsolete procedure]
`set-current-output-port!` *output-port* [obsolete procedure]
`set-notification-output-port!` *output-port* [obsolete procedure]
`set-trace-output-port!` *output-port* [obsolete procedure]
`set-interaction-i/o-port!` *i/o-port* [obsolete procedure]
These procedures are **deprecated**; instead call the corresponding parameters with an argument.

`with-input-from-port` *input-port thunk* [obsolete procedure]
`with-output-to-port` *output-port thunk* [obsolete procedure]
`with-notification-output-port` *output-port thunk* [obsolete procedure]
`with-trace-output-port` *output-port thunk* [obsolete procedure]
`with-interaction-i/o-port` *i/o-port thunk* [obsolete procedure]
These procedures are **deprecated**; instead use `parameterize` on the corresponding parameters.

`console-i/o-port` [variable]
`console-i/o-port` is an I/O port that communicates with the "console". Under unix, the console is the controlling terminal of the Scheme process. Under Windows, the console is the window that is created when Scheme starts up.

This variable is rarely used; instead programs should use one of the standard ports defined above. This variable should not be modified.

14.2 File Ports

Before Scheme can access a file for reading or writing, it is necessary to open a port to the file. This section describes procedures used to open ports to files. Such ports are closed (like any other port) by `close-port`. File ports are automatically closed if and when they are reclaimed by the garbage collector.

Before opening a file for input or output, by whatever method, the *filename* argument is converted to canonical form by calling the procedure `merge-pathnames` with *filename* as its sole argument. Thus, *filename* can be either a string or a pathname, and it is merged with the current pathname defaults to produce the pathname that is then opened.

`call-with-input-file` *filename procedure* [standard procedure]
`call-with-output-file` *filename procedure* [standard procedure]
It is an error if *procedure* does not accept one argument.

These procedures obtain a textual port obtained by opening the named file for input or output as if by `open-input-file` or `open-output-file`. The port and *procedure* are then passed to a procedure equivalent to `call-with-port`.

`call-with-binary-input-file` *filename procedure* [procedure]
`call-with-binary-output-file` *filename procedure* [procedure]
It is an error if *procedure* does not accept one argument.

These procedures obtain a binary port obtained by opening the named file for input or output as if by `open-binary-input-file` or `open-binary-output-file`. The port and *procedure* are then passed to a procedure equivalent to `call-with-port`.

`with-input-from-file` *filename thunk* [standard procedure]
`with-output-to-file` *filename thunk* [standard procedure]
The file named by *filename* is opened for input or output as if by `open-input-file` or `open-output-file`, and the new port is made to be the value returned by `current-input-port` or `current-output-port` (as used by `(read)`, `(write obj)`, and so forth). The *thunk* is then called with no arguments. When the *thunk* returns, the port is closed and the previous default is restored. It is an error if *thunk* does not accept zero arguments. Both procedures return the values yielded by *thunk*. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with `parameterize`.

`with-input-from-binary-file` *filename thunk* [obsolete procedure]
`with-output-to-binary-file` *filename thunk* [obsolete procedure]
These procedures are **deprecated**; instead use `parameterize` along with `call-with-binary-input-file` or `call-with-binary-output-file`.

`open-input-file` *filename* [procedure]
`open-binary-input-file` *filename* [procedure]
Takes a *filename* for an existing file and returns a textual input port or binary input port that is capable of delivering data from the file. If the file does not exist or cannot be opened, an error an error that satisfies `file-error?` is signaled.

`open-output-file` *filename* [*append?*] [standard procedure]
`open-binary-output-file` *filename* [*append?*] [standard procedure]
Takes a *filename* naming an output file to be created and returns a textual output port or binary output port that is capable of writing data to a new file by that name. If a file with the given name already exists, the effect is unspecified. (In that case, MIT/GNU Scheme overwrites an existing file.) If the file cannot be opened, an error that satisfies `file-error?` is signalled.

The optional argument *append?* is an MIT/GNU Scheme extension. If *append?* is given and not `#f`, the file is opened in *append* mode. In this mode, the contents of the file are not overwritten; instead any characters written to the file are appended to the end of the existing contents. If the file does not exist, append mode creates the file and writes to it in the normal way.

`open-i/o-file` *filename* [procedure]
`open-binary-i/o-file` *filename* [procedure]

Takes a *filename* referring to an existing file and returns an I/O port that is capable of both reading from and writing to the file. If the file cannot be opened, an error that satisfies `file-error?` is signalled.

This procedure is often used to open special files. For example, under unix this procedure can be used to open terminal device files, PTY device files, and named pipes.

`close-all-open-files` [procedure]

This procedure closes all file ports that are open at the time that it is called, and returns an unspecified value.

14.3 String Ports

This section describes textual input ports that read their input from given strings, and textual output ports that accumulate their output and return it as a string.

`open-input-string` *string* [*start* [*end*]] [standard procedure]

Takes a string and returns a textual input port that delivers characters from the string. If the string is modified, the effect is unspecified.

The optional arguments *start* and *end* may be used to specify that the string port delivers characters from a substring of *string*; if not given, *start* defaults to `0` and *end* defaults to `(string-length` *string*`)`.

`open-output-string` [standard procedure]

Returns a textual output port that will accumulate characters for retrieval by `get-output-string`.

`get-output-string` *port* [standard procedure]

It is an error if *port* was not created with *open-output-string*.

Returns a string consisting of the characters that have been output to the port so far in the order they were output. If the result string is modified, the effect is unspecified.

```
(parameterize ((current-output-port (open-output-string)))
  (display "piece")
  (display " by piece ")
  (display "by piece.")
  (newline)
  (get-output-string (current-output-port)))

    ⇒ "piece by piece by piece.\n"
```

`call-with-output-string` *procedure* [procedure]

The *procedure* is called with one argument, a textual output port. The values yielded by *procedure* are ignored. When *procedure* returns, `call-with-output-string` returns the port's accumulated output as a string. If the result string is modified, the effect is unspecified.

This procedure could have been defined as follows:

```
(define (call-with-output-string procedure)
  (let ((port (open-output-string)))
    (procedure port)
    (get-output-string port)))
```

`call-with-truncated-output-string` *limit procedure* [procedure]
Similar to `call-with-output-string`, except that the output is limited to at most *limit* characters. The returned value is a pair; the car of the pair is `#t` if *procedure* attempted to write more than *limit* characters, and `#f` otherwise. The cdr of the pair is a newly allocated string containing the accumulated output.

This procedure could have been defined as follows:

```
(define (call-with-truncated-output-string limit procedure)
  (let ((port (open-output-string)))
    (let ((truncated?
           (call-with-truncated-output-port limit port
                                            procedure)))
      (cons truncated? (get-output-string port)))))
```

This procedure is helpful for displaying circular lists, as shown in this example:

```
(define inf (list 'inf))
(call-with-truncated-output-string 40
  (lambda (port)
    (write inf port)))                  ⇒  (#f . "(inf)")
(set-cdr! inf inf)
(call-with-truncated-output-string 40
  (lambda (port)
    (write inf port)))
        ⇒  (#t . "(inf inf inf inf inf inf inf inf inf inf")
```

`write-to-string` *object* [*limit*] [procedure]
Writes *object* to a string output port, and returns the resulting string.

If *limit* is supplied and not `#f`, then this procedure is equivalent to the following and returns a pair instead of just a string:

```
(call-with-truncated-output-string limit
  (lambda (port)
    (write object port)))
```

`with-input-from-string` *string thunk* [obsolete procedure]
`with-output-to-string` *thunk* [obsolete procedure]
`with-output-to-truncated-string` *limit thunk* [obsolete procedure]
These procedures are **deprecated**; instead use `open-input-string`, `call-with-output-string`, or `call-with-truncated-output-string` along with `parameterize`.

14.4 Bytevector Ports

This section describes binary input ports that read their input from given bytevectors, and binary output ports that accumulate their output and return it as a bytevector.

`open-input-bytevector` *bytevector* [*start* [*end*]] [standard procedure]
Takes a bytevector and returns a binary input port that delivers bytes from the bytevector. If the bytevector is modified, the effect is unspecified.

The optional arguments *start* and *end* may be used to specify that the bytevector port delivers bytes from a portion of *bytevector*; if not given, *start* defaults to `0` and *end* defaults to `(bytevector-length` *bytevector*`)`.

`open-output-bytevector` [standard procedure]
Returns a binary output port that will accumulate bytes for retrieval by `get-output-bytevector`.

`get-output-bytevector` *port* [standard procedure]
It is an error if *port* was not created with *open-output-bytevector*.

Returns a bytevector consisting of the bytes that have been output to the port so far in the order they were output. If the result bytevector is modified, the effect is unspecified.

`call-with-output-bytevector` *procedure* [procedure]
The *procedure* is called with one argument, a binary output port. The values yielded by *procedure* are ignored. When *procedure* returns, `call-with-output-bytevector` returns the port's accumulated output as a newly allocated bytevector.

This procedure could have been defined as follows:

```
(define (call-with-output-bytevector procedure)
  (let ((port (open-output-bytevector)))
    (procedure port)
    (get-output-bytevector port)))
```

14.5 Input Procedures

This section describes the procedures that read input. Input procedures can read either from the current input port or from a given port. Remember that to read from a file, you must first open a port to the file.

Input ports can be divided into two types, called *interactive* and *non-interactive*. Interactive input ports are ports that read input from a source that is time-dependent; for example, a port that reads input from a terminal or from another program. Non-interactive input ports read input from a time-independent source, such as an ordinary file or a character string.

In this section, all optional arguments called *port* default to the current input port.

`read` [*port*] [standard procedure]
The `read` procedure converts external representations of Scheme objects into the objects themselves. It returns the next object parsable from the given textual input

port, updating *port* to point to the first character past the end of the external representation of the object.

Implementations may support extended syntax to represent record types or other types that do not have datum representations.

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 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 that satisfies `read-error?` is signaled.

The *port* remains open, and further attempts to read will also return an end-of-file object. If an end of file is encountered after the beginning of an object's written representation, but the written representation is incomplete and therefore not parsable, an error is signalled.

`read-char` [*port*] [standard procedure]
Returns the next character available from the textual input *port*, updating *port* to point to the following character. If no more characters are available, an end-of-file object is returned.

In MIT/GNU Scheme, if *port* is an interactive input port and no characters are immediately available, `read-char` will hang waiting for input, even if the port is in non-blocking mode.

`read-char-no-hang` [*port*] [procedure]
This procedure behaves exactly like `read-char` except when *port* is an interactive port in non-blocking mode, and there are no characters immediately available. In that case this procedure returns `#f` without blocking.

`unread-char` *char* [*port*] [procedure]
The given *char* must be the most-recently read character from the textual input *port*. This procedure "unreads" the character, updating *port* as if the character had never been read.

Note that this only works with characters returned by `read-char` or `read-char-no-hang`.

`peek-char` [*port*] [standard procedure]
Returns the next character available from the textual input *port*, *without* updating *port* to point to the following character. If no more characters are available, an end-of-file object is returned.

Note: 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.

`read-line` [*port*] [standard procedure]
Returns the next line of text available from the textual input *port*, updating the *port* to point to the following character. If an end of line is read, a string containing all of the text up to (but not including) the end of line is returned, and the port is updated to point just past the end of line. If an end of file is encountered before any end of line is read, but some characters have been read, a string containing those characters is returned. If an end of file is encountered before any characters are read, an end-of-file object is returned. For the purpose of this procedure, an end of line consists of either a linefeed character, a carriage return character, or a sequence of a carriage return character followed by a linefeed character. Implementations may also recognize other end of line characters or sequences.

In MIT/GNU Scheme, if *port* is an interactive input port and no characters are immediately available, `read-line` will hang waiting for input, even if the port is in non-blocking mode.

`eof-object?` *object* [standard procedure]
Returns `#t` if *object* is an end-of-file object, otherwise returns `#f`. The precise set of end-of-file objects will vary among implementations, but in any case no end-of-file object will ever be an object that can be read in using `read`.

`eof-object` [standard procedure]
Returns an end-of-file object, not necessarily unique.

`char-ready?` [*port*] [standard procedure]
Returns `#t` if a character is ready on the textual input *port* and returns `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on the given *port* is guaranteed not to hang. If the *port* is at end of file then `char-ready?` returns `#t`.

Rationale: The `char-ready?` procedure 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 ensure that characters whose existence has been asserted by `char-ready?` cannot be removed from the input. 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-string` *k* [*port*] [standard procedure]
Reads the next *k* characters, or as many as are available before the end of file, from the textual input *port* into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

Note: MIT/GNU Scheme previously defined this procedure differently, and this alternate usage is **deprecated**; please use `read-delimited-string` instead. For now, `read-string` will redirect to `read-delimited-string` as needed, but this redirection will be eliminated in a future release.

`read-string!` *string* [*port* [*start* [*end*]]] [procedure]
Reads the next end-start characters, or as many as are available before the end of file, from the textual input *port* into *string* in left-to-right order beginning at the *start* position. If *end* is not supplied, reads until the end of *string* has been reached. If

start is not supplied, reads beginning at position `0`. Returns the number of characters read. If no characters are available, an end-of-file object is returned.

In MIT/GNU Scheme, if *port* is an interactive port in non-blocking mode and no characters are immediately available, `#f` is returned without any modification of *string*.

However, if one or more characters are immediately available, the region is filled using the available characters. The procedure then returns the number of characters filled in, without waiting for further characters, even if the number of filled characters is less than the size of the region.

`read-substring!` *string start end* [*port*] [obsolete procedure]
This procedure is **deprecated**; use `read-string!` instead.

`read-u8` [*port*] [standard procedure]
Returns the next byte available from the binary input *port*, updating the *port* to point to the following byte. If no more bytes are available, an end-of-file object is returned.

In MIT/GNU Scheme, if *port* is an interactive input port in non-blocking mode and no characters are immediately available, `read-u8` will return `#f`.

`peek-u8` [*port*] [standard procedure]
Returns the next byte available from the binary input *port*, but *without* updating the *port* to point to the following byte. If no more bytes are available, an end-of-file object is returned.

In MIT/GNU Scheme, if *port* is an interactive input port in non-blocking mode and no characters are immediately available, `peek-u8` will return `#f`.

`u8-ready?` [*port*] [standard procedure]
Returns `#t` if a byte is ready on the binary input `port` and returns `#f` otherwise. If `u8-ready?` returns `#t` then the next `read-u8` operation on the given *port* is guaranteed not to hang. If the *port* is at end of file then `u8-ready?` returns `#t`.

`read-bytevector` *k* [*port*] [standard procedure]
Reads the next *k* bytes, or as many as are available before the end of file, from the binary input *port* into a newly allocated bytevector in left-to-right order and returns the bytevector. If no bytes are available before the end of file, an end-of-file object is returned.

In MIT/GNU Scheme, if *port* is an interactive input port in non-blocking mode and no characters are immediately available, `read-bytevector` will return `#f`.

However, if one or more bytes are immediately available, they are read and returned as a bytevector, without waiting for further bytes, even if the number of bytes is less than *k*.

`read-bytevector!` *bytevector* [*port* [*start* [*end*]]] [standard procedure]
Reads the next end-start bytes, or as many as are available before the end of file, from the binary input *port* into *bytevector* in left-to-right order beginning at the *start* position. If *end* is not supplied, reads until the end of *bytevector* has been reached. If *start* is not supplied, reads beginning at position `0`. Returns the number of bytes read. If no bytes are available, an end-of-file object is returned.

In MIT/GNU Scheme, if *port* is an interactive input port in non-blocking mode and no characters are immediately available, `read-bytevector!` will return `#f`.

However, if one or more bytes are immediately available, the region is filled using the available bytes. The procedure then returns the number of bytes filled in, without waiting for further bytes, even if the number of filled bytes is less than the size of the region.

`read-delimited-string` *char-set* [*port*] [procedure]

Reads characters from *port* until it finds a terminating character that is a member of *char-set* (see Section 5.3 [Character Sets], page 115) or encounters end of file. The port is updated to point to the terminating character, or to end of file if no terminating character was found. `read-delimited-string` returns the characters, up to but excluding the terminating character, as a newly allocated string.

This procedure ignores the blocking mode of the port, blocking unconditionally until it sees either a delimiter or end of file. If end of file is encountered before any characters are read, an end-of-file object is returned.

On many input ports, this operation is significantly faster than the following equivalent code using `peek-char` and `read-char`:

```
(define (read-delimited-string char-set port)
  (let ((char (peek-char port)))
    (if (eof-object? char)
        char
        (list->string
         (let loop ((char char))
           (if (or (eof-object? char)
                   (char-in-set? char char-set))
               '()
               (begin
                 (read-char port)
                 (cons char
                       (loop (peek-char port))))))))))
```

14.5.1 Reader Controls

The following parameters control the behavior of the `read` procedure.

`param:reader-radix` [parameter]

This parameter defines the radix used by the reader when it parses numbers. This is similar to passing a radix argument to `string->number`. The value of the parameter must be one of `2`, `8`, `10`, or `16`; an error is signaled if the parameter is bound to any other value.

Note that much of the number syntax is invalid for radixes other than `10`. The reader detects cases where such invalid syntax is used and signals an error. However, problems can still occur when `param:reader-radix` is bound to `16`, because syntax that normally denotes symbols can now denote numbers (e.g. `abc`). Because of this, it is usually undesirable to bind this parameter to anything other than the default.

The default value of this parameter is `10`.

`param:reader-fold-case?` [parameter]

This parameter controls whether the parser folds the case of symbols, character names, and certain other syntax. If it is bound to its default value of `#t`, symbols read by the parser are case-folded prior to being interned. Otherwise, symbols are interned without folding.

At present, it is a bad idea to use this feature, as it doesn't really make Scheme case-sensitive, and therefore can break features of the Scheme runtime that depend on case-folded symbols. Instead, use the `#!fold-case` or `#!no-fold-case` markers in your code.

`*parser-radix*` [obsolete variable]

`*parser-canonicalize-symbols?*` [obsolete variable]

These variables are **deprecated**; instead use the corresponding parameter objects.

14.6 Output Procedures

Output ports may or may not support *buffering* of output, in which output characters are collected together in a buffer and then sent to the output device all at once. (Most of the output ports implemented by the runtime system support buffering.) Sending all of the characters in the buffer to the output device is called *flushing* the buffer. In general, output procedures do not flush the buffer of an output port unless the buffer is full.

However, the standard output procedures described in this section perform what is called *discretionary* flushing of the buffer. Discretionary output flushing works as follows. After a procedure performs its output (writing characters to the output buffer), it checks to see if the port implements an operation called `discretionary-flush-output`. If so, then that operation is invoked to flush the buffer. At present, only the console port defines `discretionary-flush-output`; this is used to guarantee that output to the console appears immediately after it is written, without requiring calls to `flush-output-port`.

In this section, all optional arguments called *port* default to the current output port.

`write` *object* [*port*] [standard procedure]

Writes a representation of *object* to the given textual output *port*. Strings that appear in the written representation are enclosed in quotation marks, and within those strings backslash and quotation mark characters are escaped by backslashes. Symbols that contain non-ASCII characters are escaped with vertical lines. Character objects are written using the `#\` notation.

If *object* contains cycles which would cause an infinite loop using the normal written representation, then at least the objects that form part of the cycle must be represented using datum labels. Datum labels must not be used if there are no cycles.

Implementations may support extended syntax to represent record types or other types that do not have datum representations.

The `write` procedure returns an unspecified value.

On MIT/GNU Scheme `write` performs discretionary output flushing.

`write-shared` *object* [*port*] [standard procedure]

The `write-shared` procedure is the same as `write`, except that shared structure must be represented using datum labels for all pairs and vectors that appear more than once in the output.

`write-simple` *object* [*port*] [standard procedure]
The `write-simple` procedure is the same as `write`, except that shared structure is never represented using datum labels. This can cause `write-simple` not to terminate if *object* contains circular structure.

`display` *object* [*port*] [standard procedure]
Writes a representation of *object* to the given textual output *port*. Strings that appear in the written representation are output as if by write-string instead of by write. Symbols are not escaped. Character objects appear in the representation as if written by `write-char` instead of by `write`. The display representation of other objects is unspecified. However, `display` must not loop forever on self-referencing pairs, vectors, or records. Thus if the normal `write` representation is used, datum labels are needed to represent cycles as in `write`.

Implementations may support extended syntax to represent record types or other types that do not have datum representations.

The `display` procedure returns an unspecified value.

Rationale: The `write` procedure is intended for producing machine-readable output and `display` for producing human-readable output.

`newline` [*port*] [standard procedure]
Writes an end of line to textual output *port*. Exactly how this is done differs from one operating system to another. Returns an unspecified value.

`write-char` *char* [*port*] [standard procedure]
Writes the character *char* (not an external representation of the character) to the given textual output *port* and returns an unspecified value.

`write-string` *string* [*port* [*start* [*end*]]] [standard procedure]
Writes the characters of *string* from *start* to *end* in left-to-right order to the textual output *port*.

`write-substring` *string start end* [*port*] [obsolete procedure]
This procedure is **deprecated**; use `write-string` instead.

`write-u8` *byte* [*port*] [standard procedure]
Writes the *byte* to the given binary output *port* and returns an unspecified value.

In MIT/GNU Scheme, if *port* is an interactive output port in non-blocking mode and writing a byte would block, `write-u8` immediately returns `#f` without writing anything. Otherwise *byte* is written and `1` is returned.

`write-bytevector` *bytevector* [*port* [*start* [*end*]]] [standard procedure]
Writes the bytes of *bytevector* from *start* to *end* in left-to-right order to the binary output *port*.

In MIT/GNU Scheme, if *port* is an interactive output port in non-blocking mode `write-bytevector` will write as many bytes as it can without blocking, then returns the number of bytes written; if no bytes can be written without blocking, returns `#f` without writing anything. Otherwise `write-bytevector` returns the number of bytes actually written, which may be less than the number requested if unable to write all the bytes. (For example, if writing to a file and the file system is full.)

`flush-output-port` [*port*] [standard procedure]
Flushes any buffered output from the buffer of *port* to the underlying file or device and returns an unspecified value.

`flush-output` [*port*] [obsolete procedure]
This procedure is **deprecated**; use `flush-output-port` instead.

`fresh-line` [*port*] [procedure]
Most output ports are able to tell whether or not they are at the beginning of a line of output. If *port* is such a port, this procedure writes an end-of-line to the port only if the port is not already at the beginning of a line. If *port* is not such a port, this procedure is identical to `newline`. In either case, `fresh-line` performs discretionary output flushing and returns an unspecified value.

`write-line` *object* [*port*] [procedure]
Like `write`, except that it writes an end-of-line to *port* after writing *object*'s representation. This procedure performs discretionary output flushing and returns an unspecified value.

`beep` [*port*] [procedure]
Performs a "beep" operation on *port*, performs discretionary output flushing, and returns an unspecified value. On the console port, this usually causes the console bell to beep, but more sophisticated interactive ports may take other actions, such as flashing the screen. On most output ports, e.g. file and string output ports, this does nothing.

`clear` [*port*] [procedure]
"Clears the screen" of *port*, performs discretionary output flushing, and returns an unspecified value. On a terminal or window, this has a well-defined effect. On other output ports, e.g. file and string output ports, this does nothing.

`pp` *object* [*port* [*as-code?*]] [procedure]
`pp` prints *object* in a visually appealing and structurally revealing manner on *port*. If object is a procedure, `pp` attempts to print the source text. If the optional argument *as-code?* is true, `pp` prints lists as Scheme code, providing appropriate indentation; by default this argument is false. `pp` performs discretionary output flushing and returns an unspecified value.

The following parameters may be used with `parameterize` to change the behavior of the `write` and `display` procedures.

`param:printer-radix` [parameter]
This parameter specifies the default radix used to print numbers. Its value must be one of the exact integers `2`, `8`, `10`, or `16`; the default is `10`. For values other than `10`, numbers are prefixed to indicate their radix.

`param:printer-list-breadth-limit` [parameter]
This parameter specifies a limit on the length of the printed representation of a list or vector; for example, if the limit is `4`, only the first four elements of any list are printed, followed by ellipses to indicate any additional elements. The value of this

parameter must be an exact non-negative integer, or `#f` meaning no limit; the default is `#f`.

```
(parameterize ((param:printer-list-breadth-limit 4))
  (write-to-string '(a b c d)))
                                    ⇒ "(a b c d)"
(parameterize ((param:printer-list-breadth-limit 4))
  (write-to-string '(a b c d e)))
                                    ⇒ "(a b c d ...)"
```

`param:printer-list-depth-limit` [parameter]

This parameter specifies a limit on the nesting of lists and vectors in the printed representation. If lists (or vectors) are more deeply nested than the limit, the part of the representation that exceeds the limit is replaced by ellipses. The value of this parameter must be an exact non-negative integer, or `#f` meaning no limit; the default is `#f`.

```
(parameterize ((param:printer-list-depth-limit 4))
  (write-to-string '((((a))) b c d)))
                                    ⇒ "((((a))) b c d)"
(parameterize ((param:printer-list-depth-limit 4))
  (write-to-string '(((((a)))) b c d)))
                                    ⇒ "((((...))) b c d)"
```

`param:printer-string-length-limit` [parameter]

This parameter specifies a limit on the length of the printed representation of strings. If a string's length exceeds this limit, the part of the printed representation for the characters exceeding the limit is replaced by ellipses. The value of this parameter must be an exact non-negative integer, or `#f` meaning no limit; the default is `#f`.

```
(parameterize ((param:printer-string-length-limit 4))
  (write-to-string "abcd"))
                                    ⇒ "\"abcd\""
(parameterize ((param:printer-string-length-limit 4))
  (write-to-string "abcde"))
                                    ⇒ "\"abcd...\""
```

`param:print-with-maximum-readability?` [parameter]

This parameter, which takes a boolean value, tells the printer to use a special printed representation for objects that normally print in a form that cannot be recognized by `read`. These objects are printed using the representation `#@`*n*, where *n* is the result of calling `hash` on the object to be printed. The reader recognizes this syntax, calling `unhash` on *n* to get back the original object. Note that this printed representation can only be recognized by the Scheme program in which it was generated, because these hash numbers are different for each invocation of Scheme.

`*unparser-radix*` [obsolete variable]
`*unparser-list-breadth-limit*` [obsolete variable]
`*unparser-list-depth-limit*` [obsolete variable]
`*unparser-string-length-limit*` [obsolete variable]

`*unparse-with-maximum-readability?*` [obsolete variable]
These variables are **deprecated**; instead use the corresponding parameter objects.

14.7 Blocking Mode

An interactive port is always in one of two modes: *blocking* or *non-blocking*. This mode is independent of the terminal mode: each can be changed independently of the other. Furthermore, if it is an interactive I/O port, there are separate blocking modes for input and for output.

If an input port is in blocking mode, attempting to read from it when no input is available will cause Scheme to "block", i.e. suspend itself, until input is available. If an input port is in non-blocking mode, attempting to read from it when no input is available will cause the reading procedure to return immediately, indicating the lack of input in some way (exactly how this situation is indicated is separately specified for each procedure or operation).

An output port in blocking mode will block if the output device is not ready to accept output. In non-blocking mode it will return immediately after performing as much output as the device will allow (again, each procedure or operation reports this situation in its own way).

Interactive ports are initially in blocking mode; this can be changed at any time with the procedures defined in this section.

These procedures represent blocking mode by the symbol `blocking`, and non-blocking mode by the symbol `nonblocking`. An argument called *mode* must be one of these symbols. A *port* argument to any of these procedures may be any port, even if that port does not support blocking mode; in that case, the port is not modified in any way.

`input-port-blocking-mode` *input-port* [procedure]
`output-port-blocking-mode` *output-port* [procedure]
Returns the blocking mode of *input-port* or *output-port*. Returns `#f` if the given port doesn't support blocking mode.

`set-input-port-blocking-mode!` *input-port mode* [procedure]
`set-output-port-blocking-mode` *output-port mode* [procedure]
Changes the blocking mode of *input-port* or *output-port* to be *mode* and returns an unspecified value.

`with-input-port-blocking-mode` *input-port mode thunk* [procedure]
`with-output-port-blocking-mode` *output-port mode thunk* [procedure]
Thunk must be a procedure of no arguments.

Binds the blocking mode of *input-port* or *output-port* to be *mode*, and calls *thunk*. When *thunk* returns, the original blocking mode is restored and the values yielded by *thunk* are returned.

`port/input-blocking-mode` *input-port* [obsolete procedure]
`port/set-input-blocking-mode` *input-port mode* [obsolete procedure]
`port/with-input-blocking-mode` *input-port mode thunk* [obsolete procedure]
`port/output-blocking-mode` *output-port* [obsolete procedure]
`port/set-output-blocking-mode` *output-port mode* [obsolete procedure]

`port/with-output-blocking-mode` *output-port mode thunk* [obsolete procedure]

These procedures are **deprecated**; instead use the corresponding procedures above.

14.8 Terminal Mode

A port that reads from or writes to a terminal has a *terminal mode*; this is either *cooked* or *raw*. This mode is independent of the blocking mode: each can be changed independently of the other. Furthermore, a terminal I/O port has independent terminal modes both for input and for output.

A terminal port in cooked mode provides some standard processing to make the terminal easy to communicate with. For example, under unix, cooked mode on input reads from the terminal a line at a time and provides editing within the line, while cooked mode on output might translate linefeeds to carriage-return/linefeed pairs. In general, the precise meaning of cooked mode is operating-system dependent, and furthermore might be customizable by means of operating-system utilities. The basic idea is that cooked mode does whatever is necessary to make the terminal handle all of the usual user-interface conventions for the operating system, while keeping the program's interaction with the port as normal as possible.

A terminal port in raw mode disables all of that processing. In raw mode, characters are directly read from and written to the device without any translation or interpretation by the operating system. On input, characters are available as soon as they are typed, and are not echoed on the terminal by the operating system. In general, programs that put ports in raw mode have to know the details of interacting with the terminal. In particular, raw mode is used for writing programs such as text editors.

Terminal ports are initially in cooked mode; this can be changed at any time with the procedures defined in this section.

These procedures represent cooked mode by the symbol `cooked`, and raw mode by the symbol `raw`. An argument called *mode* must be one of these symbols. A *port* argument to any of these procedures may be any port, even if that port does not support terminal mode; in that case, the port is not modified in any way.

`input-port-terminal-mode` *input-port* [procedure]
`output-port-terminal-mode` *output-port* [procedure]

Returns the terminal mode of *input-port* or *output-port*. Returns `#f` if the given port is not a terminal port.

`set-input-port-terminal-mode!` *input-port mode* [procedure]
`set-output-port-terminal-mode!` *output-port mode* [procedure]

Changes the terminal mode of *input-port* or *output-port* to be *mode* and returns an unspecified value.

`with-input-port-terminal-mode` *input-port mode thunk* [procedure]
`with-output-port-terminal-mode` *output-port mode thunk* [procedure]

Thunk must be a procedure of no arguments.

Binds the terminal mode of *input-port* or *output-port* to be *mode*, and calls *thunk*. When *thunk* returns, the original terminal mode is restored and the values yielded by *thunk* are returned.

`port/input-terminal-mode` *input-port* [obsolete procedure]
`port/set-input-terminal-mode` *input-port mode* [obsolete procedure]
`port/with-input-terminal-mode` *input-port mode thunk* [obsolete procedure]
`port/output-terminal-mode` *output-port* [obsolete procedure]
`port/set-output-terminal-mode` *output-port mode* [obsolete procedure]
`port/with-output-terminal-mode` *output-port mode thunk* [obsolete procedure]

These procedures are **deprecated**; instead use the corresponding procedures above.

14.9 Format

The procedure `format` is very useful for producing nicely formatted text, producing good-looking messages, and so on. MIT/GNU Scheme's implementation of `format` is similar to that of Common Lisp, except that Common Lisp defines many more directives.[1]

`format` is a run-time-loadable option. To use it, execute

```
(load-option 'format)
```

once before calling it.

`format` *destination control-string argument ...* [procedure]

Writes the characters of *control-string* to *destination*, except that a tilde (`~`) introduces a *format directive*. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more *argument*s to create their output; the typical directive puts the next *argument* into the output, formatted in some special way. It is an error if no argument remains for a directive requiring an argument, but it is not an error if one or more arguments remain unprocessed by a directive.

The output is sent to *destination*. If *destination* is `#f`, a string is created that contains the output; this string is returned as the value of the call to `format`. In all other cases `format` returns an unspecified value. If *destination* is `#t`, the output is sent to the current output port. Otherwise, *destination* must be an output port, and the output is sent there.

This procedure performs discretionary output flushing (see Section 14.6 [Output Procedures], page 238).

A `format` directive consists of a tilde (`~`), optional prefix parameters separated by commas, optional colon (`:`) and at-sign (`@`) modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the directive character is ignored. The prefix parameters are generally integers, notated as optionally signed decimal numbers. If both the colon and at-sign modifiers are given, they may appear in either order.

In place of a prefix parameter to a directive, you can put the letter '`V`' (or '`v`'), which takes an *argument* for use as a parameter to the directive. Normally this should be an exact integer. This feature allows variable-width fields and the like. You can also use the character '`#`' in place of a parameter; it represents the number of arguments remaining to be processed.

[1] This description of `format` is adapted from *Common Lisp, The Language*, second edition, section 22.3.3.

It is an error to give a format directive more parameters than it is described here as accepting. It is also an error to give colon or at-sign modifiers to a directive in a combination not specifically described here as being meaningful.

`~A` The next *argument*, which may be any object, is printed as if by `display`. `~`*mincol*`A` inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The `@` modifier causes the spaces to be inserted on the left rather than the right.

`~S` The next *argument*, which may be any object, is printed as if by `write`. `~`*mincol*`S` inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The `@` modifier causes the spaces to be inserted on the left rather than the right.

`~%` This outputs a `#\newline` character. `~`*n*`%` outputs *n* newlines. No *argument* is used. Simply putting a newline in *control-string* would work, but `~%` is often used because it makes the control string look nicer in the middle of a program.

`~~` This outputs a tilde. `~`*n*`~` outputs *n* tildes.

`~`*newline* Tilde immediately followed by a newline ignores the newline and any following non-newline whitespace characters. With an `@`, the newline is left in place, but any following whitespace is ignored. This directive is typically used when *control-string* is too long to fit nicely into one line of the program:

```
(define (type-clash-error procedure arg spec actual)
  (format
   #t
   "~%Procedure ~S~%requires its %A argument ~
    to be of type ~S,~%but it was called with ~
    an argument of type ~S.~%"
   procedure arg spec actual))

(type-clash-error 'vector-ref
                  "first"
                  'integer
                  'vector)
```

prints

```
Procedure vector-ref
requires its first argument to be of type integer,
but it was called with an argument of type vector.
```

Note that in this example newlines appear in the output only as specified by the `~%` directives; the actual newline characters in the control string are suppressed because each is preceded by a tilde.

14.10 Custom Output

MIT/GNU Scheme provides hooks for specifying that specified objects have special written representations. There are no restrictions on the written representations.

`define-print-method` *predicate print-method* [procedure]
Defines the print method for objects satisfying *predicate* to be *print-method*. The *predicate* argument must be a unary procedure that returns true for the objects to print specially, and *print-method* must be a binary procedure that accepts one of those objects and a textual output port.

Although *print-method* can print the object in any way, we strongly recomment using one of the following special printers.

`standard-print-method` *name* [*get-parts*] [procedure]
The *name* argument may be a unary procedure, a string, or a symbol; if it is a procedure it is called with the object to be printed as its argument and should return a string or a symbol. The *get-parts* argument, if provided, must be a unary procedure that is called with the object to be printed and must return a list of objects. If *get-parts* is not provided, it defaults to a procedure that returns an empty list.

The output generated by this method is in a standard format:

```
#[<name> <hash> <part>...]
```

where `<name>` is the string or symbol from *name* as printed by `display`, `<hash>` is a unique nonnegative integer generated by calling `hash-object` on the object, and the `<part>`s are the result of calling *get-parts* as printed by `write` and separated by spaces.

One significant advantage of print methods generated by `standard-print-method` is that the parts returned by *get-parts* are examined when searching for circular structure (as by `write`) or shared structure (as by `write-shared`). In effect the printer sees one of these objects as a compound object containing those parts.

`bracketed-print-method` *name printer* [procedure]
The *name* argument may be a unary procedure, a string, or a symbol; if it is a procedure it is called with the object to be printed as its argument and should return a string or a symbol. The *printer* argument must be a binary procedure, which is called with the object to print and a textual output port as its arguments.

Similar to `standard-print-method`, this procedure prints an object

```
#[<name> <hash><output>]
```

where `<name>` is the string or symbol from *name* as printed by `display`, `<hash>` is a unique nonnegative integer generated by calling `hash-object` on the object, and `<output>` is the text written by *printer*.

This procedure has the benefit of printing objects using the standard bracketed form, but because its *output* is unstructured can not be examined for sharing or circularity. Generally speaking it's preferable to use `standard-print-method` instead.

The following are deprecated procedures that have been replaced by the above.

`set-record-type-unparser-method!` *record-type unparser-method* [obsolete procedure]

This procedure is **deprecated**; instead use

```
(define-print-method (record-predicate record-type)
  unparser-method)
```

provided that *unparser-method* is really a print method.

`unparser/set-tagged-vector-method!` *tag unparser-method* [obsolete procedure]

`unparser/set-tagged-pair-method!` *tag unparser-method* [obsolete procedure]

These procedures arg **deprecated**. There is no direct replacement for them.

These were primarily used by `define-structure`, which now generates new-style print methods. If you have other uses of these, it should be possible to translate them to use `define-print-method` with hand-written predicates.

`standard-unparser-method` *name procedure* [obsolete procedure]

This procedure is **deprecated**; it is currently an alias for `bracketed-print-method`.

`with-current-unparser-state` *unparser-state procedure* [obsolete procedure]

This procedure is **deprecated**, with no direct replacement. In general just use *procedure* without wrapping it.

14.11 Prompting

This section describes procedures that prompt the user for input. Why should the programmer use these procedures when it is possible to do prompting using ordinary input and output procedures? One reason is that the prompting procedures are more succinct. However, a second and better reason is that the prompting procedures can be separately customized for each user interface, providing more natural interaction. The interfaces for Edwin and for GNU Emacs have already been customized in this fashion; because Edwin and Emacs are very similar editors, their customizations provide very similar behavior.

Each of these procedure accepts an optional argument called *port*, which if given must be an I/O port. If not given, this port defaults to the value of `(interaction-i/o-port)`; this is initially the console I/O port.

`prompt-for-command-expression` *prompt* [*port*] [procedure]

Prompts the user for an expression that is to be executed as a command. This is the procedure called by the REP loop to read the user's expressions.

If *prompt* is a string, it is used verbatim as the prompt string. Otherwise, it must be a pair whose car is the symbol '`standard`' and whose cdr is a string; in this case the prompt string is formed by prepending to the string the current REP loop "level number" and a space. Also, a space is appended to the string, unless it already ends in a space or is an empty string.

The default behavior of this procedure is to print a fresh line, a newline, and the prompt string; flush the output buffer; then read an object and return it. v Under Edwin and Emacs, before the object is read, the interaction buffer is put into a mode that allows expressions to be edited and submitted for input using specific editor

commands. The first expression that is submitted is returned as the value of this procedure.

`prompt-for-command-char` *prompt* [*port*] [procedure]
Prompts the user for a single character that is to be executed as a command; the returned character is guaranteed to satisfy `char-graphic?`. If at all possible, the character is read from the user interface using a mode that reads the character as a single keystroke; in other words, it should not be necessary for the user to follow the character with a carriage return or something similar.

This is the procedure called by `debug` and `where` to read the user's commands.

If *prompt* is a string, it is used verbatim as the prompt string. Otherwise, it must be a pair whose car is `standard` and whose cdr is a string; in this case the prompt string is formed by prepending to the string the current REP loop "level number" and a space. Also, a space is appended to the string, unless it already ends in a space or is an empty string.

The default behavior of this procedure is to print a fresh line, a newline, and the prompt string; flush the output buffer; read a character in raw mode, echo that character, and return it.

Under Edwin and Emacs, instead of reading a character, the interaction buffer is put into a mode in which graphic characters submit themselves as input. After this mode change, the first such character submitted is returned as the value of this procedure.

`prompt-for-expression` *prompt* [*port*] [procedure]
Prompts the user for an expression.

The prompt string is formed by appending a colon and a space to *prompt*, unless *prompt* already ends in a space or is the null string.

The default behavior of this procedure is to print a fresh line, a newline, and the prompt string; flush the output buffer; then read an object and return it.

Under Edwin and Emacs, the expression is read in the minibuffer.

`prompt-for-evaluated-expression` *prompt* [*environment* [*port*]] [procedure]
Prompts the user for an evaluated expression. Calls `prompt-for-expression` to read an expression, then evaluates the expression using *environment*; if *environment* is not given, the REP loop environment is used.

`prompt-for-confirmation` *prompt* [*port*] [procedure]
Prompts the user for confirmation. The result yielded by this procedure is a boolean.

The prompt string is formed by appending the string `" (y or n)? "` to *prompt*, unless *prompt* already ends in a space or is the null string.

The default behavior of this procedure is to print a fresh line, a newline, and the prompt string; flush the output buffer; then read a character in raw mode. If the character is `#\y`, `#\Y`, or `#\space`, the procedure returns `#t`; If the character is `#\n`, `#\N`, or `#\rubout`, the procedure returns `#f`. Otherwise the prompt is repeated.

Under Edwin or Emacs, the confirmation is read in the minibuffer.

14.12 Textual Port Primitives

This section describes the low-level operations that can be used to build and manipulate textual I/O ports. The purpose of these operations is to allow programmers to construct new kinds of textual I/O ports.

The mechanisms described in this section are exclusively for textual ports; binary ports can't be customized. In this section, any reference to a "port" that isn't modified by "textual" or "binary" is assumed to be a textual port.

The abstract model of a textual I/O port, as implemented here, is a combination of a set of named operations and a state. The state is an arbitrary object, the meaning of which is determined by the operations. The operations are defined by a mapping from names to procedures.

The set of named operations is represented by an object called a *textual port type*. A port type is constructed from a set of named operations, and is subsequently used to construct a port. The port type completely specifies the behavior of the port. Port types also support a simple form of inheritance, allowing you to create new ports that are similar to existing ports.

The port operations are divided into two classes:

Standard operations
: There is a specific set of standard operations for input ports, and a different set for output ports. Applications can assume that the standard input operations are implemented for all input ports, and likewise the standard output operations are implemented for all output ports.

Custom operations
: Some ports support additional operations. For example, ports that implement output to terminals (or windows) may define an operation named `y-size` that returns the height of the terminal in characters. Because only some ports will implement these operations, programs that use custom operations must test each port for their existence, and be prepared to deal with ports that do not implement them.

14.12.1 Textual Port Types

The procedures in this section provide means for constructing port types with standard and custom operations, and accessing their operations.

`make-textual-port-type` *operations port-type* [procedure]
: Creates and returns a new port type. *Operations* must be a list; each element is a list of two elements, the name of the operation (a symbol) and the procedure that implements it. *Port-type* is either `#f` or a port type; if it is a port type, any operations implemented by *port-type* but not specified in *operations* will be implemented by the resulting port type.

  Operations need not contain definitions for all of the standard operations; the procedure will provide defaults for any standard operations that are not defined. At a minimum, the following operations must be defined: for input ports, `read-char` and `peek-char`; for output ports, either `write-char` or `write-substring`. I/O ports must supply the minimum operations for both input and output.

If an operation in *operations* is defined to be `#f`, then the corresponding operation in *port-type* is *not* inherited.

If `read-char` is defined in *operations*, then any standard input operations defined in *port-type* are ignored. Likewise, if `write-char` or `write-substring` is defined in *operations*, then any standard output operations defined in *port-type* are ignored. This feature allows overriding the standard operations without having to enumerate them.

`textual-port-type?` *object* [procedure]
`textual-input-port-type?` *object* [procedure]
`textual-output-port-type?` *object* [procedure]
`textual-i/o-port-type?` *object* [procedure]
These predicates return `#t` if *object* is a port type, input-port type, output-port type, or I/O-port type, respectively. Otherwise, they return `#f`.

`make-port-type` *operations port-type* [obsolete procedure]
`port-type?` *object* [obsolete procedure]
`input-port-type?` *object* [obsolete procedure]
`output-port-type?` *object* [obsolete procedure]
`i/o-port-type?` *object* [obsolete procedure]
These procedures are **deprecated**; use the procedures defined above.

`port-type/operations` *port-type* [obsolete procedure]
`port-type/operation-names` *port-type* [obsolete procedure]
`port-type/operation` *port-type symbol* [obsolete procedure]
These procedures are **deprecated** and will be removed in the near future. There are no replacements planned.

14.12.2 Constructors and Accessors for Textual Ports

The procedures in this section provide means for constructing ports, accessing the type of a port, and manipulating the state of a port.

`make-textual-port` *port-type state* [procedure]
Returns a new port with type *port-type* and the given *state*. The port will be an input, output, or I/O port according to *port-type*.

`textual-port-type` *textual-port* [procedure]
Returns the port type of *textual-port*.

`textual-port-state` *textual-port* [procedure]
Returns the state component of *textual-port*.

`set-textual-port-state!` *textual-port object* [procedure]
Changes the state component of *textual-port* to be *object*. Returns an unspecified value.

`textual-port-operation` *textual-port symbol* [procedure]
Returns the operation named *symbol* for *textual-port*. If *textual-port* has no such operation, returns `#f`.

`textual-port-operation-names` *textual-port* [procedure]
Returns a newly allocated list whose elements are the names of the operations implemented by *textual-port*.

`make-port` *port-type state* [obsolete procedure]
`port/type` *textual-port* [obsolete procedure]
`port/state` *textual-port* [obsolete procedure]
`set-port/state!` *textual-port object* [obsolete procedure]
`port/operation` *textual-port symbol* [obsolete procedure]
`port/operation-names` *port* [obsolete procedure]
These procedures are **deprecated**; use the procedures defined above.

14.12.3 Textual Input Port Operations

This section describes the standard operations on textual input ports. Following that, some useful custom operations are described.

`read-char` *port* [operation on `textual input port`]
Removes the next character available from *port* and returns it. If *port* has no more characters and will never have any (e.g. at the end of an input file), this operation returns an end-of-file object. If *port* has no more characters but will eventually have some more (e.g. a terminal where nothing has been typed recently), and it is in non-blocking mode, `#f` is returned; otherwise the operation hangs until input is available.

`peek-char` *port* [operation on `textual input port`]
Reads the next character available from *port* and returns it. The character is *not* removed from *port*, and a subsequent attempt to read from the port will get that character again. In other respects this operation behaves like `read-char`.

`char-ready?` *port k* [operation on `textual input port`]
`char-ready?` returns `#t` if at least one character is available to be read from *port*. If no characters are available, the operation waits up to *k* milliseconds before returning `#f`, returning immediately if any characters become available while it is waiting.

`read-string` *port char-set* [operation on `textual input port`]
`discard-chars` *port char-set* [operation on `textual input port`]
These operations are like `read-char`, except that they read or discard multiple characters at once. All characters up to, but excluding, the first character in *char-set* (or end of file) are read from *port*. `read-string` returns these characters as a newly allocated string, while `discard-chars` discards them and returns an unspecified value. These operations hang until sufficient input is available, even if *port* is in non-blocking mode. If end of file is encountered before any input characters, `read-string` returns an end-of-file object.

`read-substring` *port string start end* [operation on `textual input port`]
Reads characters from *port* into the substring defined by *string*, *start*, and *end* until either the substring has been filled or there are no more characters available. Returns the number of characters written to the substring.

If *port* is an interactive port, and at least one character is immediately available, the available characters are written to the substring and this operation returns immediately. If no characters are available, and *port* is in blocking mode, the operation blocks until at least one character is available. Otherwise, the operation returns `#f` immediately.

This is an extremely fast way to read characters from a port.

`input-port/read-char` *textual-input-port* [procedure]
`input-port/peek-char` *textual-input-port* [procedure]
`input-port/char-ready?` *textual-input-port k* [procedure]
`input-port/read-string` *textual-input-port char-set* [procedure]
`input-port/discard-chars` *textual-input-port char-set* [procedure]
`input-port/read-substring` *textual-input-port string start end* [procedure]
Each of these procedures invokes the respective operation on *textual-input-port*. For example, the following are equivalent:

```
(input-port/read-char textual-input-port)
((textual-port-operation textual-input-port 'read-char)
 textual-input-port)
```

The following custom operations are implemented for input ports to files, and will also work with some other kinds of input ports:

`eof?` *port* [operation on `textual input port`]
Returns `#t` if *port* is known to be at end of file, otherwise it returns `#f`.

`chars-remaining` *port* [operation on `textual input port`]
Returns an estimate of the number of characters remaining to be read from *port*. This is useful only when *port* is a file port in binary mode; in other cases, it returns `#f`.

`buffered-input-chars` *port* [operation on `textual input port`]
Returns the number of unread characters that are stored in *port*'s buffer. This will always be less than or equal to the buffer's size.

`input-buffer-size` *port* [operation on `textual input port`]
Returns the maximum number of characters that *port*'s buffer can hold.

`set-input-buffer-size` *port size* [operation on `textual input port`]
Resizes *port*'s buffer so that it can hold at most *size* characters. Characters in the buffer are discarded. *Size* must be an exact non-negative integer.

14.12.4 Textual Output Port Operations

This section describes the standard operations on output ports. Following that, some useful custom operations are described.

`write-char` *port char* [operation on `textual output port`]
Writes *char* to *port* and returns an unspecified value.

`write-substring` *port string start end* [operation on `textual output port`]
Writes the substring specified by *string*, *start*, and *end* to *port* and returns an unspecified value. Equivalent to writing the characters of the substring, one by one, to *port*, but is implemented very efficiently.

`fresh-line` *port* [operation on `textual output port`]
Most output ports are able to tell whether or not they are at the beginning of a line of output. If *port* is such a port, end-of-line is written to the port only if the port is not already at the beginning of a line. If *port* is not such a port, an end-of-line is unconditionally written to the port. Returns an unspecified value.

`flush-output` *port* [operation on `textual output port`]
If *port* is buffered, this causes its buffer to be written out. Otherwise it has no effect. Returns an unspecified value.

`discretionary-flush-output` *port* [operation on `textual output port`]
Normally, this operation does nothing. However, ports that support discretionary output flushing implement this operation identically to `flush-output`.

`output-port/write-char` *textual-output-port char* [procedure]
`output-port/write-substring` *textual-output-port string start end* [procedure]
`output-port/fresh-line` *textual-output-port* [procedure]
`output-port/flush-output` *textual-output-port* [procedure]
`output-port/discretionary-flush-output` *textual-output-port* [procedure]
Each of these procedures invokes the respective operation on *textual-output-port*. For example, the following are equivalent:

```
(output-port/write-char textual-output-port char)
((textual-port-operation textual-output-port 'write-char)
 textual-output-port char)
```

`output-port/write-string` *textual-output-port string* [procedure]
Writes *string* to *textual-output-port*. Equivalent to

```
(output-port/write-substring textual-output-port
                             string
                             0
                             (string-length string))
```

The following custom operations are generally useful.

`buffered-output-chars` *port* [operation on `textual output port`]
Returns the number of unwritten characters that are stored in *port*'s buffer. This will always be less than or equal to the buffer's size.

`output-buffer-size` *port* [operation on `textual output port`]
Returns the maximum number of characters that *port*'s buffer can hold.

`set-output-buffer-size` *port size* [operation on `textual output port`]
Resizes *port*'s buffer so that it can hold at most *size* characters. Characters in the buffer are discarded. *Size* must be an exact non-negative integer.

`x-size` *port* [operation on `textual output port`]
Returns an exact positive integer that is the width of *port* in characters. If *port* has no natural width, e.g. if it is a file port, `#f` is returned.

`y-size` *port* [operation on `textual output port`]
Returns an exact positive integer that is the height of *port* in characters. If *port* has no natural height, e.g. if it is a file port, `#f` is returned.

`output-port/x-size` *textual-output-port* [procedure]
This procedure invokes the custom operation whose name is the symbol `x-size`, if it exists. If the `x-size` operation is both defined and returns a value other than `#f`, that value is returned as the result of this procedure. Otherwise, `output-port/x-size` returns a default value (currently `80`).

`output-port/x-size` is useful for programs that tailor their output to the width of the display (a fairly common practice). If the output device is not a display, such programs normally want some reasonable default width to work with, and this procedure provides exactly that.

`output-port/y-size` *textual-output-port* [procedure]
This procedure invokes the custom operation whose name is the symbol `y-size`, if it exists. If the `y-size` operation is defined, the value it returns is returned as the result of this procedure; otherwise, `#f` is returned.

14.13 Parser Buffers

The *parser buffer* mechanism facilitates construction of parsers for complex grammars. It does this by providing an input stream with unbounded buffering and backtracking. The amount of buffering is under program control. The stream can backtrack to any position in the buffer.

The mechanism defines two data types: the *parser buffer* and the *parser-buffer pointer*. A parser buffer is like an input port with buffering and backtracking. A parser-buffer pointer is a pointer into the stream of characters provided by a parser buffer.

Note that all of the procedures defined here consider a parser buffer to contain a stream of Unicode characters.

There are several constructors for parser buffers:

`textual-input-port->parser-buffer` *textual-input-port* [procedure]
`input-port->parser-buffer` *textual-input-port* [obsolete procedure]
Returns a parser buffer that buffers characters read from *textual-input-port*.

`substring->parser-buffer` *string start end* [procedure]
Returns a parser buffer that buffers the characters in the argument substring. This is equivalent to creating a string input port and calling `textual-input-port->parser-buffer`, but it runs faster and uses less memory.

`string->parser-buffer` *string* [procedure]
Like `substring->parser-buffer` but buffers the entire string.

`source->parser-buffer` *source* [procedure]
: Returns a parser buffer that buffers the characters returned by calling *source*. *Source* is a procedure of three arguments: a string, a start index, and an end index (in other words, a substring specifier). Each time *source* is called, it writes some characters in the substring, and returns the number of characters written. When there are no more characters available, it returns zero. It must not return zero in any other circumstance.

Parser buffers and parser-buffer pointers may be distinguished from other objects:

`parser-buffer?` *object* [procedure]
: Returns `#t` if *object* is a parser buffer, otherwise returns `#f`.

`parser-buffer-pointer?` *object* [procedure]
: Returns `#t` if *object* is a parser-buffer pointer, otherwise returns `#f`.

Characters can be read from a parser buffer much as they can be read from an input port. The parser buffer maintains an internal pointer indicating its current position in the input stream. Additionally, the buffer remembers all characters that were previously read, and can look at characters arbitrarily far ahead in the stream. It is this buffering capability that facilitates complex matching and backtracking.

`read-parser-buffer-char` *buffer* [procedure]
: Returns the next character in *buffer*, advancing the internal pointer past that character. If there are no more characters available, returns `#f` and leaves the internal pointer unchanged.

`peek-parser-buffer-char` *buffer* [procedure]
: Returns the next character in *buffer*, or `#f` if no characters are available. Leaves the internal pointer unchanged.

`parser-buffer-ref` *buffer index* [procedure]
: Returns a character in *buffer*. *Index* is a non-negative integer specifying the character to be returned. If *index* is zero, returns the next available character; if it is one, returns the character after that, and so on. If *index* specifies a position after the last character in *buffer*, returns `#f`. Leaves the internal pointer unchanged.

The internal pointer of a parser buffer can be read or written:

`get-parser-buffer-pointer` *buffer* [procedure]
: Returns a parser-buffer pointer object corresponding to the internal pointer of *buffer*.

`set-parser-buffer-pointer!` *buffer pointer* [procedure]
: Sets the internal pointer of *buffer* to the position specified by *pointer*. *Pointer* must have been returned from a previous call of `get-parser-buffer-pointer` on *buffer*. Additionally, if some of *buffer*'s characters have been discarded by `discard-parser-buffer-head!`, *pointer* must be outside the range that was discarded.

`get-parser-buffer-tail` *buffer pointer* [procedure]
: Returns a newly-allocated string consisting of all of the characters in *buffer* that fall between *pointer* and *buffer*'s internal pointer. *Pointer* must have been returned from

a previous call of `get-parser-buffer-pointer` on *buffer*. Additionally, if some of *buffer*'s characters have been discarded by `discard-parser-buffer-head!`, *pointer* must be outside the range that was discarded.

`discard-parser-buffer-head!` *buffer* [procedure]
Discards all characters in *buffer* that have already been read; in other words, all characters prior to the internal pointer. After this operation has completed, it is no longer possible to move the internal pointer backwards past the current position by calling `set-parser-buffer-pointer!`.

The next rather large set of procedures does conditional matching against the contents of a parser buffer. All matching is performed relative to the buffer's internal pointer, so the first character to be matched against is the next character that would be returned by `peek-parser-buffer-char`. The returned value is always `#t` for a successful match, and `#f` otherwise. For procedures whose names do not end in '`-no-advance`', a successful match also moves the internal pointer of the buffer forward to the end of the matched text; otherwise the internal pointer is unchanged.

`match-parser-buffer-char` *buffer char* [procedure]
`match-parser-buffer-char-ci` *buffer char* [procedure]
`match-parser-buffer-not-char` *buffer char* [procedure]
`match-parser-buffer-not-char-ci` *buffer char* [procedure]
`match-parser-buffer-char-no-advance` *buffer char* [procedure]
`match-parser-buffer-char-ci-no-advance` *buffer char* [procedure]
`match-parser-buffer-not-char-no-advance` *buffer char* [procedure]
`match-parser-buffer-not-char-ci-no-advance` *buffer char* [procedure]
Each of these procedures compares a single character in *buffer* to *char*. The basic comparison `match-parser-buffer-char` compares the character to *char* using `char=?`. The procedures whose names contain the '`-ci`' modifier do case-insensitive comparison (i.e. they use `char-ci=?`). The procedures whose names contain the '`not-`' modifier are successful if the character *doesn't* match *char*.

`match-parser-buffer-char-in-set` *buffer char-set* [procedure]
`match-parser-buffer-char-in-set-no-advance` *buffer char-set* [procedure]
These procedures compare the next character in *buffer* against *char-set* using `char-in-set?`.

`match-parser-buffer-string` *buffer string* [procedure]
`match-parser-buffer-string-ci` *buffer string* [procedure]
`match-parser-buffer-string-no-advance` *buffer string* [procedure]
`match-parser-buffer-string-ci-no-advance` *buffer string* [procedure]
These procedures match *string* against *buffer*'s contents. The '`-ci`' procedures do case-insensitive matching.

`match-parser-buffer-substring` *buffer string start end* [procedure]
`match-parser-buffer-substring-ci` *buffer string start end* [procedure]
`match-parser-buffer-substring-no-advance` *buffer string start end* [procedure]

`match-parser-buffer-substring-ci-no-advance` *buffer string start end* [procedure]

These procedures match the specified substring against *buffer*'s contents. The '`-ci`' procedures do case-insensitive matching.

The remaining procedures provide information that can be used to identify locations in a parser buffer's stream.

`parser-buffer-position-string` *pointer* [procedure]

Returns a string describing the location of *pointer* in terms of its character and line indexes. This resulting string is meant to be presented to an end user in order to direct their attention to a feature in the input stream. In this string, the indexes are presented as one-based numbers.

Pointer may alternatively be a parser buffer, in which case it is equivalent to having specified the buffer's internal pointer.

`parser-buffer-pointer-index` *pointer* [procedure]
`parser-buffer-pointer-line` *pointer* [procedure]

Returns the character or line index, respectively, of *pointer*. Both indexes are zero-based.

14.14 Parser Language

Although it is possible to write parsers using the parser-buffer abstraction (see Section 14.13 [Parser Buffers], page 254), it is tedious. The problem is that the abstraction isn't closely matched to the way that people think about syntactic structures. In this section, we introduce a higher-level mechanism that greatly simplifies the implementation of a parser.

The *parser language* described here allows the programmer to write BNF-like specifications that are translated into efficient Scheme code at compile time. The language is declarative, but it can be freely mixed with Scheme code; this allows the parsing of grammars that aren't conveniently described in the language.

The language also provides backtracking. For example, this expression matches any sequence of alphanumeric characters followed by a single alphabetic character:

```
(*matcher
 (seq (* (char-set char-set:alphanumeric))
      (char-set char-set:alphabetic)))
```

The way that this works is that the matcher matches alphanumeric characters in the input stream until it finds a non-alphanumeric character. It then tries to match an alphabetic character, which of course fails. At this point, if it matched at least one alphanumeric character, it *backtracks*: the last matched alphanumeric is "unmatched", and it again attempts to match an alphabetic character. The backtracking can be arbitrarily deep; the matcher will continue to back up until it finds a way to match the remainder of the expression.

So far, this sounds a lot like regular-expression matching (see Section 6.2 [Regular Expressions], page 132). However, there are some important differences.

- The parser language uses a Scheme-like syntax that is easier to read and write than regular-expression notation.

- The language provides macros so that common syntactic constructs can be abstracted.
- The language mixes easily with Scheme code, allowing the full power of Scheme to be applied to program around limitations in the parser language.
- The language provides expressive facilities for converting syntax into parsed structure. It also makes it easy to convert parsed strings into meaningful objects (e.g. numbers).
- The language is compiled into machine language; regular expressions are usually interpreted.

Here is an example that shows off several of the features of the parser language. The example is a parser for XML start tags:

```
(*parser
 (with-pointer p
   (seq "<"
        parse-name
        parse-attribute-list
        (alt (match ">")
             (match "/>")
             (sexp
              (lambda (b)
                (error
                 (string-append
                  "Unterminated start tag at "
                  (parser-buffer-position-string p)))))))))
```

This shows that the basic description of a start tag is very similar to its BNF. Non-terminal symbols `parse-name` and `parse-attribute-list` do most of the work, and the noise strings `"<"` and `">"` are the syntactic markers delimiting the form. There are two alternate endings for start tags, and if the parser doesn't find either of the endings, the Scheme code (wrapped in `sexp`) is run to signal an error. The error procedure `perror` takes a pointer `p`, which it uses to indicate the position in the input stream at which the error occurred. In this case, that is the beginning of the start tag, i.e. the position of the leading `"<"` marker.

This example still looks pretty complicated, mostly due to the error-signalling code. In practice, this is abstracted into a macro, after which the expression is quite succinct:

```
(*parser
 (bracket "start tag"
     (seq (noise (string "<")) parse-name)
     (match (alt (string ">") (string "/>")))
   parse-attribute-list))
```

The `bracket` macro captures the pattern of a bracketed item, and hides much of the detail.

The parser language actually consists of two languages: one for defining matchers, and one for defining parsers. The languages are intentionally very similar, and are meant to be used together. Each sub-language is described below in its own section.

The parser language is a run-time-loadable option; to use it, execute

```
(load-option '*parser)
```

once before compiling any code that uses the language.

14.14.1 *Matcher

The *matcher language* is a declarative language for specifying a *matcher procedure*. A matcher procedure is a procedure that accepts a single parser-buffer argument and returns a boolean value indicating whether the match it performs was successful. If the match succeeds, the internal pointer of the parser buffer is moved forward over the matched text. If the match fails, the internal pointer is unchanged.

For example, here is a matcher procedure that matches the character '`a`':

```
(lambda (b) (match-parser-buffer-char b #\a))
```

Here is another example that matches two given characters, *c1* and *c2*, in sequence:

```
(lambda (b)
  (let ((p (get-parser-buffer-pointer b)))
    (if (match-parser-buffer-char b c1)
        (if (match-parser-buffer-char b c2)
            #t
            (begin
              (set-parser-buffer-pointer! b p)
              #f))
        #f)))
```

This is code is clear, but has lots of details that get in the way of understanding what it is doing. Here is the same example in the matcher language:

```
(*matcher (seq (char c1) (char c2)))
```

This is much simpler and more intuitive. And it generates virtually the same code:

```
(pp (*matcher (seq (char c1) (char c2))))
⊣ (lambda (#[b1])
⊣   (let ((#[p1] (get-parser-buffer-pointer #[b1])))
⊣     (and (match-parser-buffer-char #[b1] c1)
⊣          (if (match-parser-buffer-char #[b1] c2)
⊣              #t
⊣              (begin
⊣                (set-parser-buffer-pointer! #[b1] #[p1])
⊣                #f)))))
```

Now that we have seen an example of the language, it's time to look at the detail. The `*matcher` special form is the interface between the matcher language and Scheme.

`*matcher` *mexp* [special form]

The operand *mexp* is an expression in the matcher language. The `*matcher` expression expands into Scheme code that implements a matcher procedure.

Here are the predefined matcher expressions. New matcher expressions can be defined using the macro facility (see Section 14.14.3 [Parser-language Macros], page 265). We will start with the primitive expressions.

`char` *expression* [matcher expression]
`char-ci` *expression* [matcher expression]
`not-char` *expression* [matcher expression]

`not-char-ci` *expression* [matcher expression]

These expressions match a given character. In each case, the *expression* operand is a Scheme expression that must evaluate to a character at run time. The '`-ci`' expressions do case-insensitive matching. The '`not-`' expressions match any character other than the given one.

`string` *expression* [matcher expression]
`string-ci` *expression* [matcher expression]

These expressions match a given string. The *expression* operand is a Scheme expression that must evaluate to a string at run time. The `string-ci` expression does case-insensitive matching.

`char-set` *expression* [matcher expression]

These expressions match a single character that is a member of a given character set. The *expression* operand is a Scheme expression that must evaluate to a character set at run time.

`end-of-input` [matcher expression]

The `end-of-input` expression is successful only when there are no more characters available to be matched.

`discard-matched` [matcher expression]

The `discard-matched` expression always successfully matches the null string. However, it isn't meant to be used as a matching expression; it is used for its effect. `discard-matched` causes all of the buffered text prior to this point to be discarded (i.e. it calls `discard-parser-buffer-head!` on the parser buffer).

Note that `discard-matched` may not be used in certain places in a matcher expression. The reason for this is that it deliberately discards information needed for backtracking, so it may not be used in a place where subsequent backtracking will need to back over it. As a rule of thumb, use `discard-matched` only in the last operand of a `seq` or `alt` expression (including any `seq` or `alt` expressions in which it is indirectly contained).

In addition to the above primitive expressions, there are two convenient abbreviations. A character literal (e.g. '`#\A`') is a legal primitive expression, and is equivalent to a `char` expression with that literal as its operand (e.g. '`(char #\A)`'). Likewise, a string literal is equivalent to a `string` expression (e.g. '`(string "abc")`').

Next there are several combinator expressions. These closely correspond to similar combinators in regular expressions. Parameters named *mexp* are arbitrary expressions in the matcher language.

`seq` *mexp* ... [matcher expression]

This matches each *mexp* operand in sequence. For example,

```
(seq (char-set char-set:alphabetic)
     (char-set char-set:numeric))
```

matches an alphabetic character followed by a numeric character, such as '`H4`'.

Note that if there are no *mexp* operands, the `seq` expression successfully matches the null string.

`alt` *mexp ...* [matcher expression]

This attempts to match each *mexp* operand in order from left to right. The first one that successfully matches becomes the match for the entire `alt` expression.

The `alt` expression participates in backtracking. If one of the *mexp* operands matches, but the overall match in which this expression is embedded fails, the backtracking mechanism will cause the `alt` expression to try the remaining *mexp* operands. For example, if the expression

```
(seq (alt "ab" "a") "b")
```

is matched against the text '`abc`', the `alt` expression will initially match its first operand. But it will then fail to match the second operand of the `seq` expression. This will cause the `alt` to be restarted, at which time it will match '`a`', and the overall match will succeed.

Note that if there are no *mexp* operands, the `alt` match will always fail.

`*` *mexp* [matcher expression]

This matches zero or more occurrences of the *mexp* operand. (Consequently this match always succeeds.)

The `*` expression participates in backtracking; if it matches *N* occurrences of *mexp*, but the overall match fails, it will backtrack to *N-1* occurrences and continue. If the overall match continues to fail, the `*` expression will continue to backtrack until there are no occurrences left.

`+` *mexp* [matcher expression]

This matches one or more occurrences of the *mexp* operand. It is equivalent to

```
(seq mexp (* mexp))
```

`?` *mexp* [matcher expression]

This matches zero or one occurrences of the *mexp* operand. It is equivalent to

```
(alt mexp (seq))
```

`sexp` *expression* [matcher expression]

The `sexp` expression allows arbitrary Scheme code to be embedded inside a matcher. The *expression* operand must evaluate to a matcher procedure at run time; the procedure is called to match the parser buffer. For example,

```
(*matcher
 (seq "a"
      (sexp parse-foo)
      "b"))
```

expands to

```
(lambda (#[b1])
  (let ((#[p1] (get-parser-buffer-pointer #[b1])))
    (and (match-parser-buffer-char #[b1] #\a)
         (if (parse-foo #[b1])
             (if (match-parser-buffer-char #[b1] #\b)
                 #t
                 (begin
                   (set-parser-buffer-pointer! #[b1] #[p1])
                   #f))
             (begin
               (set-parser-buffer-pointer! #[b1] #[p1])
               #f)))))
```

The case in which *expression* is a symbol is so common that it has an abbreviation: '`(sexp` *`symbol`*`)`' may be abbreviated as just *symbol*.

`with-pointer` *identifier mexp* [matcher expression]

The `with-pointer` expression fetches the parser buffer's internal pointer (using `get-parser-buffer-pointer`), binds it to *identifier*, and then matches the pattern specified by *mexp*. *Identifier* must be a symbol.

This is meant to be used on conjunction with `sexp`, as a way to capture a pointer to a part of the input stream that is outside the `sexp` expression. An example of the use of `with-pointer` appears above (see [with-pointer example], page 258).

14.14.2 *Parser

The *parser language* is a declarative language for specifying a *parser procedure*. A parser procedure is a procedure that accepts a single parser-buffer argument and parses some of the input from the buffer. If the parse is successful, the procedure returns a vector of objects that are the result of the parse, and the internal pointer of the parser buffer is advanced past the input that was parsed. If the parse fails, the procedure returns `#f` and the internal pointer is unchanged. This interface is much like that of a matcher procedure, except that on success the parser procedure returns a vector of values rather than `#t`.

The `*parser` special form is the interface between the parser language and Scheme.

`*parser` *pexp* [special form]

The operand *pexp* is an expression in the parser language. The `*parser` expression expands into Scheme code that implements a parser procedure.

There are several primitive expressions in the parser language. The first two provide a bridge to the matcher language (see Section 14.14.1 [*Matcher], page 259):

`match` *mexp* [parser expression]

The `match` expression performs a match on the parser buffer. The match to be performed is specified by *mexp*, which is an expression in the matcher language. If the match is successful, the result of the `match` expression is a vector of one element: a string containing that text.

`noise` *mexp* [parser expression]

The `noise` expression performs a match on the parser buffer. The match to be performed is specified by *mexp*, which is an expression in the matcher language. If

the match is successful, the result of the `noise` expression is a vector of zero elements. (In other words, the text is matched and then thrown away.)

The *mexp* operand is often a known character or string, so in the case that *mexp* is a character or string literal, the `noise` expression can be abbreviated as the literal. In other words, '`(noise "foo")`' can be abbreviated just '`"foo"`'.

`values` *expression ...* [parser expression]

Sometimes it is useful to be able to insert arbitrary values into the parser result. The `values` expression supports this. The *expression* arguments are arbitrary Scheme expressions that are evaluated at run time and returned in a vector. The `values` expression always succeeds and never modifies the internal pointer of the parser buffer.

`discard-matched` [parser expression]

The `discard-matched` expression always succeeds, returning a vector of zero elements. In all other respects it is identical to the `discard-matched` expression in the matcher language.

Next there are several combinator expressions. Parameters named *pexp* are arbitrary expressions in the parser language. The first few combinators are direct equivalents of those in the matcher language.

`seq` *pexp ...* [parser expression]

The `seq` expression parses each of the *pexp* operands in order. If all of the *pexp* operands successfully match, the result is the concatenation of their values (by `vector-append`).

`alt` *pexp ...* [parser expression]

The `alt` expression attempts to parse each *pexp* operand in order from left to right. The first one that successfully parses produces the result for the entire `alt` expression.

Like the `alt` expression in the matcher language, this expression participates in backtracking.

`*` *pexp* [parser expression]

The `*` expression parses zero or more occurrences of *pexp*. The results of the parsed occurrences are concatenated together (by `vector-append`) to produce the expression's result.

Like the `*` expression in the matcher language, this expression participates in backtracking.

`+` *pexp* [parser expression]

The `*` expression parses one or more occurrences of *pexp*. It is equivalent to

```
(seq pexp (* pexp))
```

`?` *pexp* [parser expression]

The `*` expression parses zero or one occurrences of *pexp*. It is equivalent to

```
(alt pexp (seq))
```

The next three expressions do not have equivalents in the matcher language. Each accepts a single *pexp* argument, which is parsed in the usual way. These expressions perform transformations on the returned values of a successful match.

`transform` *expression pexp* [parser expression]

The `transform` expression performs an arbitrary transformation of the values returned by parsing *pexp*. *Expression* is a Scheme expression that must evaluate to a procedure at run time. If *pexp* is successfully parsed, the procedure is called with the vector of values as its argument, and must return a vector or `#f`. If it returns a vector, the parse is successful, and those are the resulting values. If it returns `#f`, the parse fails and the internal pointer of the parser buffer is returned to what it was before *pexp* was parsed.

For example:

```
(transform (lambda (v) (if (= 0 (vector-length v)) #f v)) ...)
```

`encapsulate` *expression pexp* [parser expression]

The `encapsulate` expression transforms the values returned by parsing *pexp* into a single value. *Expression* is a Scheme expression that must evaluate to a procedure at run time. If *pexp* is successfully parsed, the procedure is called with the vector of values as its argument, and may return any Scheme object. The result of the `encapsulate` expression is a vector of length one containing that object. (And consequently `encapsulate` doesn't change the success or failure of *pexp*, only its value.)

For example:

```
(encapsulate vector->list ...)
```

`map` *expression pexp* [parser expression]

The `map` expression performs a per-element transform on the values returned by parsing *pexp*. *Expression* is a Scheme expression that must evaluate to a procedure at run time. If *pexp* is successfully parsed, the procedure is mapped (by `vector-map`) over the values returned from the parse. The mapped values are returned as the result of the `map` expression. (And consequently `map` doesn't change the success or failure of *pexp*, nor the number of values returned.)

For example:

```
(map string->symbol ...)
```

Finally, as in the matcher language, we have `sexp` and `with-pointer` to support embedding Scheme code in the parser.

`sexp` *expression* [parser expression]

The `sexp` expression allows arbitrary Scheme code to be embedded inside a parser. The *expression* operand must evaluate to a parser procedure at run time; the procedure is called to parse the parser buffer. This is the parser-language equivalent of the `sexp` expression in the matcher language.

The case in which *expression* is a symbol is so common that it has an abbreviation: '`(sexp` *`symbol`*`)`' may be abbreviated as just *symbol*.

`with-pointer` *identifier pexp* [parser expression]

The `with-pointer` expression fetches the parser buffer's internal pointer (using `get-parser-buffer-pointer`), binds it to *identifier*, and then parses the pattern specified by *pexp*. *Identifier* must be a symbol. This is the parser-language equivalent of the `with-pointer` expression in the matcher language.

14.14.3 Parser-language Macros

The parser and matcher languages provide a macro facility so that common patterns can be abstracted. The macro facility allows new expression types to be independently defined in the two languages. The macros are defined in hierarchically organized tables, so that different applications can have private macro bindings.

`define-*matcher-macro` *formals expression* [special form]
`define-*parser-macro` *formals expression* [special form]

These special forms are used to define macros in the matcher and parser language, respectively. *Formals* is like the *formals* list of a `define` special form, and *expression* is a Scheme expression.

If *formals* is a list (or improper list) of symbols, the first symbol in the list is the name of the macro, and the remaining symbols are interpreted as the *formals* of a lambda expression. A lambda expression is formed by combining the latter *formals* with the *expression*, and this lambda expression, when evaluated, becomes the *expander*. The defined macro accepts the same number of operands as the expander. A macro instance is expanded by applying the expander to the list of operands; the result of the application is interpreted as a replacement expression for the macro instance.

If *formals* is a symbol, it is the name of the macro. In this case, the expander is a procedure of no arguments whose body is *expression*. When the *formals* symbol appears by itself as an expression in the language, the expander is called with no arguments, and the result is interpreted as a replacement expression for the symbol.

`define-*matcher-expander` *identifier expander* [procedure]
`define-*parser-expander` *identifier expander* [procedure]

These procedures provide a procedural interface to the macro-definition mechanism. *Identifier* must be a symbol, and *expander* must be an expander procedure, as defined above. Instances of the `define-*matcher-macro` and `define-*parser-macro` special forms expand into calls to these procedures.

The remaining procedures define the interface to the parser-macros table abstraction. Each parser-macro table has a separate binding space for macros in the matcher and parser languages. However, the table inherits bindings from one specified table; it's not possible to inherit matcher-language bindings from one table and parser-language bindings from another.

`make-parser-macros` *parent-table* [procedure]

Create and return a new parser-macro table that inherits from *parent-table*. *Parent-table* must be either a parser-macro table, or `#f`; usually it is specified as the value of `global-parser-macros`.

`parser-macros?` *object* [procedure]

This is a predicate for parser-macro tables.

`global-parser-macros` [procedure]

Return the global parser-macro table. This table is predefined and contains all of the bindings documented here.

There is a "current" table at all times, and macro definitions are always placed in this table. By default, the current table is the global macro table, but the following procedures allow this to be changed.

`current-parser-macros` [procedure]
Return the current parser-macro table.

`set-current-parser-macros!` *table* [procedure]
Change the current parser-macro table to *table*, which must satisfy `parser-macros?`.

`with-current-parser-macros` *table thunk* [procedure]
Bind the current parser-macro table to *table*, call *thunk* with no arguments, then restore the original table binding. The value returned by *thunk* is the returned as the value of this procedure. *Table* must satisfy `parser-macros?`, and *thunk* must be a procedure of no arguments.

14.15 XML Support

MIT/GNU Scheme provides a simple non-validating XML parser. This parser is believed to be conformant with XML 1.0. It passes all of the tests in the "xmltest" directory of the XML conformance tests (dated 2001-03-15). The parser supports XML namespaces; it doesn't support external document type declarations (DTDs), and it doesn't yet support XML 1.1. The output of the parser is a record tree that closely reflects the structure of the XML document.

MIT/GNU Scheme also provides support for writing an XML record tree to an output port. There is no guarantee that parsing an XML document and writing it back out will make a verbatim copy of the document. The output will be semantically identical but may have small syntactic differences. For example, entities are substituted during the parsing process.

The purpose of the XML support is to provide a mechanism for reading and writing simple XML documents. In the future this support may be further developed to support a standard interface such as DOM or SAX.

The XML support is a run-time-loadable option; to use it, execute

```
(load-option 'xml)
```

once before running any code that uses it.

14.15.1 XML Input

The primary entry point for the XML parser is `read-xml`, which reads characters from a port and returns an XML document record. The character coding of the input is determined by reading some of the input stream and looking for a byte order mark and/or an encoding in the XML declaration. We support all ISO 8859 codings, as well as UTF-8, UTF-16, and UTF-32.

When an XHTML document is read, the parser provides entity definitions for all of the named XHTML characters; for example, it defines '` `' and '`©`'. In order for a document to be recognized as XHTML, it must contain an XHTML DTD, such as this:

```
<!DOCTYPE html
          PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

At present the parser recognizes XHTML Strict 1.0 and XHTML 1.1 documents.

`read-xml` *port* [*pi-handlers*] [procedure]
Read an XML document from *port* and return the corresponding XML document record.

Pi-handlers, if specified, must be an association list. Each element of *pi-handlers* must be a list of two elements: a symbol and a procedure. When the parser encounters processing instructions with a name that appears in *pi-handlers*, the procedure is called with one argument, which is the text of the processing instructions. The procedure must return a list of XML structure records that are legal for the context of the processing instructions.

`read-xml-file` *pathname* [*pi-handlers*] [procedure]
This convenience procedure simplifies reading XML from a file. It is roughly equivalent to

```
(define (read-xml-file pathname #!optional pi-handlers)
  (call-with-input-file pathname
    (lambda (port)
      (read-xml port pi-handlers))))
```

`string->xml` *string* [*start* [*end* [*pi-handlers*]]] [procedure]
This convenience procedure simplifies reading XML from a string. The *string* argument may be a string or a wide string. It is roughly equivalent to

```
(define (string->xml string #!optional start end pi-handlers)
  (read-xml (open-input-string string start end)
            pi-handlers))
```

14.15.2 XML Output

The following procedures serialize XML document records into character sequences. All are virtually identical except for the way that the character sequence is represented.

Each procedure will accept either an `xml-document` record or any of the other XML record types. This makes it possible to write fragments of XML documents, although you should keep in mind that such fragments aren't documents and won't generally be accepted by any XML parser.

If the *xml* being written is an `xml-document` record, the procedures `write-xml` and `write-xml-file` will look for a contained `xml-declaration` record and its `encoding` attribute. If the `encoding` is a supported value, the output will be encoded as specified; otherwise it will be encoded as UTF-8.

When an XHTML document record is written, named XHTML characters are translated into their corresponding entities. For example, the character '`#\U+00A0`' is written as '` `'. In order for an XML document record to be recognized as XHTML, it must have a DTD record that satisfies the predicate `html-dtd?`.

`write-xml` *xml port* [procedure]
Write *xml* to *port*. Note that character encoding will only be done if *port* supports it.

`write-xml-file` *xml pathname* [procedure]

Write *xml* to the file specified by *pathname*. Roughly equivalent to

```
(define (write-xml-file xml pathname)
  (call-with-output-file pathname
    (lambda (port)
      (write-xml xml port))))
```

`xml->wide-string` *xml* [procedure]

Convert *xml* to a wide string. No character encoding is used, since wide strings can represent all characters without encoding. Roughly equivalent to

```
(define (xml->wide-string xml)
  (call-with-wide-output-string
   (lambda (port)
     (write-xml xml port))))
```

`xml->string` *xml* [procedure]

Convert *xml* to a character string encoded as UTF-8. Roughly equivalent to

```
(define (xml->string xml)
  (wide-string->utf8-string (xml->wide-string xml)))
```

14.15.3 XML Names

MIT/GNU Scheme implements XML names in a slightly complex way. Unfortunately, this complexity is a direct consequence of the definition of XML names rather than a mis-feature of this implementation.

The reason that XML names are complex is that XML namespace support, which was added after XML was standardized, is not very well integrated with the core XML definition. The most obvious problem is that names can't have associated namespaces when they appear in the DTD of a document, even if the body of the document uses them. Consequently, it must be possible to compare non-associated names with associated names.

An XML name consists of two parts: the *qname*, which is a symbol, possibly including a namespace prefix; and the *Uniform Resource Identifier* (URI), which identifies an optional namespace.

`make-xml-name` *qname uri* [procedure]

Creates and returns an XML name. *Qname* must be a symbol whose name satisfies `string-is-xml-name?`. *Uri* must satisfy either `absolute-uri?` or `null-xml-namespace-uri?`. The returned value is an XML name that satisfies `xml-name?`.

If *uri* is the *null* namespace (satisfies `null-xml-namespace-uri?`), the returned value is a symbol equivalent to *qname*. This means that an ordinary symbol can be used as an XML name when there is no namespace associated with the name.

For convenience, *qname* may be a string, in which case it is converted to a symbol using `make-xml-qname`.

For convenience, *uri* may be any object that `->uri` is able to convert to a URI record, provided the resulting URI meets the above restrictions.

`xml-name?` *object* [procedure]

Returns `#t` if *object* is an XML name, and `#f` otherwise.

`xml-name->symbol` *xml-name* [procedure]
Returns the symbol part of *xml-name*.

`xml-name-uri` *xml-name* [procedure]
Returns the *URI* of *xml-name*. The result always satisfies `absolute-uri?` or `null-xml-namespace-uri?`.

`xml-name-string` *xml-name* [procedure]
Returns the *qname* of *xml-name* as a string. Equivalent to

```
(symbol->string (xml-name->symbol xml-name))
```

The next two procedures get the *prefix* and *local part* of an XML name, respectively. The prefix of an XML name is the part of the qname to the left of the colon, while the local part is the part of the qname to the right of the colon. If there is no colon in the qname, the local part is the entire qname, and the prefix is the null symbol (i.e. '`||`').

`xml-name-prefix` *xml-name* [procedure]
Returns the *prefix* of *xml-name* as a symbol.

`xml-name-local` *xml-name* [procedure]
Returns the *local part* of *xml-name* as a symbol.

The next procedure compares two XML names for equality. The rules for equality are slightly complex, in order to permit comparing names in the DTD with names in the document body. So, if both of the names have non-null namespace URIs, then the names are equal if and only if their local parts are equal and their URIs are equal. (The prefixes of the names are not considered in this case.) Otherwise, the names are equal if and only if their qnames are equal.

`xml-name=?` *xml-name-1 xml-name-2* [procedure]
Returns `#t` if *xml-name-1* and *xml-name-2* are the same name, and `#f` otherwise.

These next procedures define the data abstraction for qnames. While qnames are represented as symbols, only symbols whose names satisfy `string-is-xml-name?` are qnames.

`make-xml-qname` *string* [procedure]
String must satisfy `string-is-xml-name?`. Returns the qname corresponding to *string* (the symbol whose name is *string*).

`xml-qname?` *object* [procedure]
Returns `#t` if *object* is a qname, otherwise returns `#f`.

`xml-qname-prefix` *qname* [procedure]
Returns the prefix of *qname* as a symbol.

`xml-qname-local` *qname* [procedure]
Returns the local part of *qname* as a symbol.

The prefix of a qname or XML name may be absent if there is no colon in the name. The absent, or null, prefix is abstracted by the next two procedures. Note that the null prefix is a symbol, just like non-null prefixes.

`null-xml-name-prefix` [procedure]
Returns the null prefix.

`null-xml-name-prefix?` *object* [procedure]
Returns `#t` if *object* is the null prefix, otherwise returns `#f`.

The namespace URI of an XML name may be null, meaning that there is no namespace associated with the name. This namespace is represented by a relative URI record whose string representation is the null string.

`null-xml-namespace-uri` [procedure]
Returns the null namespace URI record.

`null-xml-namespace-uri?` *object* [procedure]
Returns `#t` if *object* is the null namespace URI record, otherwise returns `#f`.

The following values are two distinguished URI records.

`xml-uri` [variable]
`xml-uri` is the URI reserved for use by the XML recommendation. This URI must be used with the '`xml`' prefix.

`xmlns-uri` [variable]
`xmlns-uri` is the URI reserved for use by the XML namespace recommendation. This URI must be used with the '`xmlns`' prefix.

`make-xml-nmtoken` *string* [procedure]

`xml-nmtoken?` *object* [procedure]

`string-is-xml-name?` *string* [procedure]

`string-is-xml-nmtoken?` *string* [procedure]

14.15.4 XML Structure

The output from the XML parser and the input to the XML output procedure is a complex data structure composed of a hierarchy of typed components. Each component is a record whose fields correspond to parts of the XML structure that the record represents. There are no special operations on these records; each is a tuple with named subparts. The root record type is `xml-document`, which represents a complete XML document.

Each record type *type* has the following associated bindings:

`<type>` is a variable bound to the record-type descriptor for *type*. The record-type descriptor may be used as a specializer in SOS method definitions, which greatly simplifies code to dispatch on these types.

type`?` is a predicate for records of type *type*. It accepts one argument, which can be any object, and returns `#t` if the object is a record of this type, or `#f` otherwise.

`make-`*type*
is a constructor for records of type *type*. It accepts one argument for each field of *type*, in the same order that they are written in the type description, and returns a newly-allocated record of that type.

type-field
: is an accessor procedure for the field *field* in records of type *type*. It accepts one argument, which must be a record of that type, and returns the contents of the corresponding field in the record.

set-*type-field*!
: is a modifier procedure for the field *field* in records of type *type*. It accepts two arguments: the first must be a record of that type, and the second is a new value for the corresponding field. The record's field is modified to have the new value.

`xml-document` *declaration misc-1 dtd misc-2 root misc-3* [record type]
: The `xml-document` record is the top-level record representing a complete XML document. *Declaration* is either an `xml-declaration` object or `#f`. *Dtd* is either an `xml-dtd` object or `#f`. *Root* is an `xml-element` object. *Misc-1*, *misc-2*, and *misc-3* are lists of miscellaneous items; a miscellaneous item is either an `xml-comment` object, an `xml-processing-instructions` object, or a string of whitespace.

`xml-declaration` *version encoding standalone* [record type]
: The `xml-declaration` record represents the '`<?xml ... ?>`' declaration that optionally appears at the beginning of an XML document. *Version* is a version string, typically `"1.0"`. *Encoding* is either an encoding string or `#f`. *Standalone* is either `"yes"`, `"no"`, or `#f`.

`xml-element` *name attributes contents* [record type]
: The `xml-element` record represents general XML elements; the bulk of a typical XML document consists of these elements. *Name* is the element name (an XML name). *Attributes* is a list of XML attribute objects. *Contents* is a list of the contents of the element. Each element of this list is either a string, an `xml-element` record or an `xml-processing-instructions` record.

`xml-processing-instructions` *name text* [record type]
: The `xml-processing-instructions` record represents processing instructions, which have the form '`<?`*name* `... ?>`'. These instructions are intended to contain non-XML data that will be processed by another interpreter; for example they might contain PHP programs. The *name* field is the processor name (a symbol), and the *text* field is the body of the instructions (a string).

`xml-dtd` *root external internal* [record type]
: The `xml-dtd` record represents a document type declaration. The *root* field is an XML name for the root element of the document. *External* is either an `xml-external-id` record or `#f`. *Internal* is a list of DTD element records (e.g. `xml-!element`, `xml-!attlist`, etc.).

The remaining record types are valid only within a DTD.

`xml-!element` *name content-type* [record type]
: The `xml-!element` record represents an element-type declaration. *Name* is the XML name of the type being declared (a symbol). *Content-type* describes the type and can have several different values, as follows:

- The XML names '`EMPTY`' and '`ANY`' correspond to the XML keywords of the same name.
- A list '`(MIX` *`type ...`*`)`' corresponds to the '`(#PCDATA | type | ...)`' syntax.

`xml-!attlist` *name definitions* [record type]
The `xml-!attlist` record represents an attribute-list declaration. *Name* is the XML name of the type for which attributes are being declared (a symbol). *Definitions* is a list of attribute definitions, each of which is a list of three elements `(`*`name type default`*`)`. *Name* is an XML name for the name of the attribute (a symbol). *Type* describes the attribute type, and can have one of the following values:

- The XML names '`CDATA`', '`IDREFS`', '`IDREF`', '`ID`', '`ENTITY`', '`ENTITIES`', '`NMTOKENS`', and '`NMTOKEN`' correspond to the XML keywords of the same names.
- A list '`(NOTATION` *`name1 name2 ...`*`)`' corresponds to the '`NOTATION (`*`name1 | name2 ...`*`)`' syntax.
- A list '`(ENUMERATED` *`name1 name2 ...`*`)`' corresponds to the '`(`*`name1 | name2 ...`*`)`' syntax.

Default describes the default value for the attribute, and can have one of the following values:

- The XML names '`#REQUIRED`' and '`#IMPLIED`' correspond to the XML keywords of the same names.
- A list '`(#FIXED` *`value`*`)`' corresponds to the '`#FIXED "`*`value`*`"`' syntax. *Value* is represented as a string.
- A list '`(DEFAULT` *`value`*`)`' corresponds to the '`"`*`value`*`"`' syntax. *Value* is represented as a string.

`xml-!entity` *name value* [record type]
The `xml-!entity` record represents a general entity declaration. *Name* is an XML name for the entity. *Value* is the entity's value, either a string or an `xml-external-id` record.

`xml-parameter-!entity` *name value* [record type]
The `xml-parameter-!entity` record represents a parameter entity declaration. *Name* is an XML name for the entity. *Value* is the entity's value, either a string or an `xml-external-id` record.

`xml-unparsed-!entity` *name id notation* [record type]
The `xml-unparsed-!entity` record represents an unparsed entity declaration. `Name` is an XML name for the entity. *Id* is an `xml-external-id` record. *Notation* is an XML name for the notation.

`xml-!notation` *name id* [record type]
The `xml-!notation` record represents a notation declaration. `Name` is an XML name for the notation. *Id* is an `xml-external-id` record.

`xml-external-id` *id uri* [record type]
The `xml-external-id` record is a reference to an external DTD. This reference consists of two parts: *id* is a public ID literal, corresponding to the '`PUBLIC`' keyword,

while *uri* is a system literal, corresponding to the '`SYSTEM`' keyword. Either or both may be present, depending on the context. *Id* is represented as a string, while *uri* is represented as a URI record.

15 Operating-System Interface

The Scheme standard provides a simple mechanism for reading and writing files: file ports. MIT/GNU Scheme provides additional tools for dealing with other aspects of the operating system:

- *Pathnames* are a reasonably operating-system independent tool for manipulating the component parts of file names. This can be useful for implementing defaulting of file name components.
- Control over the *current working directory*: the place in the file system from which relative file names are interpreted.
- Procedures that rename, copy, delete, and test for the existence of files. Also, procedures that return detailed information about a particular file, such as its type (directory, link, etc.) or length.
- Procedures for reading the contents of a directory.
- Procedures for obtaining times in various formats, converting between the formats, and generating human-readable time strings.
- Procedures to run other programs as subprocesses of Scheme, to read their output, and write input to them.
- A means to determine the operating system Scheme is running under.

15.1 Pathnames

MIT/GNU Scheme programs need to use names to designate files. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, here is a table of several file systems (actually, operating systems that provide file systems) and what equivalent file names might look like for each one:

```
System          File Name
------          ---------
TOPS-20         <LISPIO>FORMAT.FASL.13
TOPS-10         FORMAT.FAS[1,4]
ITS             LISPIO;FORMAT FASL
MULTICS         >udd>LispIO>format.fasl
TENEX           <LISPIO>FORMAT.FASL;13
VAX/VMS         [LISPIO]FORMAT.FAS;13
UNIX            /usr/lispio/format.fasl
DOS             C:\USR\LISPIO\FORMAT.FAS
```

It would be impossible for each program that deals with file names to know about each different file name format that exists; a new operating system to which Scheme was ported might use a format different from any of its predecessors. Therefore, MIT/GNU Scheme provides *two* ways to represent file names: *filenames* (also called *namestrings*), which are strings in the implementation-dependent form customary for the file system, and *pathnames*, which are special abstract data objects that represent file names in an implementation-independent way. Procedures are provided to convert between these two representations, and all manipulations of files can be expressed in machine-independent terms by using pathnames.

In order to allow MIT/GNU Scheme programs to operate in a network environment that may have more than one kind of file system, the pathname facility allows a file name to specify which file system is to be used. In this context, each file system is called a *host*, in keeping with the usual networking terminology.[1]

Note that the examples given in this section are specific to unix pathnames. Pathnames for other operating systems have different external representations.

15.1.1 Filenames and Pathnames

Pathname objects are usually created by parsing filenames (character strings) into component parts. MIT/GNU Scheme provides operations that convert filenames into pathnames and vice versa.

`->pathname` *object* [procedure]

Returns a pathname that is the equivalent of *object*. *Object* must be a pathname or a string. If *object* is a pathname, it is returned. If *object* is a string, this procedure returns the pathname that corresponds to the string; in this case it is equivalent to `(parse-namestring` *object* `#f #f)`.

```
(->pathname "foo")          ⇒  #[pathname 65 "foo"]
(->pathname "/usr/morris")  ⇒  #[pathname 66 "/usr/morris"]
```

`parse-namestring` *thing* [*host* [*defaults*]] [procedure]

This turns *thing* into a pathname. *Thing* must be a pathname or a string. If *thing* is a pathname, it is returned. If *thing* is a string, this procedure returns the pathname that corresponds to the string, parsed according to the syntax of the file system specified by *host*.

This procedure *does not* do defaulting of pathname components.

The optional arguments are used to determine what syntax should be used for parsing the string. In general this is only really useful if your implementation of MIT/GNU Scheme supports more than one file system, otherwise you would use `->pathname`. If given, *host* must be a host object or `#f`, and *defaults* must be a pathname. *Host* specifies the syntax used to parse the string. If *host* is not given or `#f`, the host component from *defaults* is used instead; if *defaults* is not given, the host component from `param:default-pathname-defaults` is used.

`->namestring` *pathname* [procedure]

`->namestring` returns a newly allocated string that is the filename corresponding to *pathname*.

```
(->namestring (->pathname "/usr/morris/minor.van"))
     ⇒  "/usr/morris/minor.van"
```

`pathname-simplify` *pathname* [procedure]

Returns a pathname that locates the same file or directory as *pathname*, but is in some sense simpler. Note that `pathname-simplify` might not always be able to simplify the pathname, e.g. on unix with symbolic links the directory `/usr/morris/../` need not

[1] This introduction is adapted from *Common Lisp, The Language*, second edition, section 23.1.

be the same as `/usr/`. In cases of uncertainty the behavior is conservative, returning the original or a partly simplified pathname.

```
(pathname-simplify "/usr/morris/../morris/dance")
     ⇒  #[pathname "/usr/morris/dance"]
```

15.1.2 Components of Pathnames

A pathname object always has six components, described below. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the Scheme implementation.

host The name of the file system on which the file resides. In the current implementation, this component is always a host object that is filled in automatically by the runtime system. When specifying the host component, use either `#f` or the value of the variable `local-host`.

device Corresponds to the "device" or "file structure" concept in many host file systems: the name of a (logical or physical) device containing files. This component is the drive letter for PC file systems, and is unused for unix file systems.

directory Corresponds to the "directory" concept in many host file systems: the name of a group of related files (typically those belonging to a single user or project). This component is always used for all file systems.

name The name of a group of files that can be thought of as conceptually the "same" file. This component is always used for all file systems.

type Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is. Files with the same name but different type are usually related in some specific way, such as one being a source file, another the compiled form of that source, and a third the listing of error messages from the compiler. This component is currently used for all file systems, and is formed by taking the characters that follow the last dot in the namestring.

version Corresponds to the "version number" concept in many host file systems. Typically this is a number that is incremented every time the file is modified. This component is currently unused for all file systems.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a specification (possibly only a partial specification) of how to access a file. A pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with a version of `newest` may refer to the same file as a pathname with the same components except a certain number as the version. Indeed, a pathname with version `newest` may refer to different files as time passes, because the meaning of such a pathname depends on the state of the file system. In file systems with such facilities as "links", multiple file names, logical devices, and so on, two pathnames that look quite different may turn out to address the same file. To access a file given a pathname, one must do a file-system operation such as `open-input-file`.

Two important operations involving pathnames are *parsing* and *merging*. Parsing is the conversion of a filename (which might be something supplied interactively by the users

when asked to supply the name of a file) into a pathname object. This operation is implementation-dependent, because the format of filenames is implementation-dependent. Merging takes a pathname with missing components and supplies values for those components from a source of default values.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is `#f`. Before the file system interface can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in. Pathnames with missing components are used internally for various purposes; in particular, parsing a namestring that does not specify certain components will result in a pathname with missing components.

Any component of a pathname may be the symbol `unspecific`, meaning that the component simply does not exist, for file systems in which such a value makes no sense. For example, unix and Windows file systems usually do not support version numbers, so the version component for such a host might be `unspecific`.[2]

In addition to `#f` and `unspecific`, the components of a pathname may take on the following meaningful values:

host An implementation-defined type which may be tested for using the `host?` predicate.

device On systems that support this component (Windows), it may be specified as a string containing a single alphabetic character, for which the alphabetic case is ignored.

directory A non-empty list, which represents a *directory path*: a sequence of directories, each of which has a name in the previous directory, the last of which is the directory specified by the entire path. Each element in such a path specifies the name of the directory relative to the directory specified by the elements to its left. The first element of the list is either the symbol `absolute` or the symbol `relative`. If the first element in the list is the symbol `absolute`, then the directory component (and subsequently the pathname) is *absolute*; the first component in the sequence is to be found at the "root" of the file system. If the directory is *relative* then the first component is to be found in some as yet unspecified directory; typically this is later specified to be the *current working directory*.

Aside from `absolute` and `relative`, which may only appear as the first element of the list, each subsequent element in the list is either: a string, which is a literal component; the symbol `wild`, meaningful only when used in conjunction with the directory reader; or the symbol `up`, meaning the next directory is the "parent" of the previous one. `up` corresponds to the file `..` in unix and PC file systems.

(The following note does not refer to any file system currently supported by MIT/GNU Scheme, but is included for completeness.) In file systems that do not have "hierarchical" structure, a specified directory component will always be a list whose first element is `absolute`. If the system does not support directories other than a single global directory, the list will have no other elements. If

[2] This description is adapted from *Common Lisp, The Language*, second edition, section 23.1.1.

the system supports "flat" directories, i.e. a global set of directories with no subdirectories, then the list will contain a second element, which is either a string or `wild`. In other words, a non-hierarchical file system is treated as if it were hierarchical, but the hierarchical features are unused. This representation is somewhat inconvenient for such file systems, but it discourages programmers from making code depend on the lack of a file hierarchy.

name A string, which is a literal component; or the symbol `wild`, meaningful only when used in conjunction with the directory reader.

type A string, which is a literal component; or the symbol `wild`, meaningful only when used in conjunction with the directory reader.

version An exact positive integer, which is a literal component; the symbol `newest`, which means to choose the largest available version number for that file; the symbol `oldest`, which means to choose the smallest version number; or the symbol `wild`, meaningful only when used in conjunction with the directory reader. In the future some other possible values may be added, e.g. `installed`. Note that currently no file systems support version numbers; thus this component is not used and should be specified as `#f`.

`make-pathname` *host device directory name type version* [procedure]
Returns a pathname object whose components are the respective arguments. Each argument must satisfy the restrictions for the corresponding component, which were outlined above.

```
(make-pathname #f
               #f
               '(absolute "usr" "morris")
               "foo"
               "scm"
               #f)
     ⇒  #[pathname 67 "/usr/morris/foo.scm"]
```

`pathname-host` *pathname* [procedure]
`pathname-device` *pathname* [procedure]
`pathname-directory` *pathname* [procedure]
`pathname-name` *pathname* [procedure]
`pathname-type` *pathname* [procedure]
`pathname-version` *pathname* [procedure]
Returns a particular component of *pathname*.

```
(define x (->pathname "/usr/morris/foo.scm"))
(pathname-host x)       ⇒  #[host 1]
(pathname-device x)     ⇒  unspecific
(pathname-directory x)  ⇒  (absolute "usr" "morris")
(pathname-name x)       ⇒  "foo"
(pathname-type x)       ⇒  "scm"
(pathname-version x)    ⇒  unspecific
```

`pathname-new-device` *pathname device* [procedure]
`pathname-new-directory` *pathname directory* [procedure]

`pathname-new-name` *pathname name* [procedure]
`pathname-new-type` *pathname type* [procedure]
`pathname-new-version` *pathname version* [procedure]
Returns a new copy of *pathname* with the respective component replaced by the second argument. *Pathname* is unchanged. Portable programs should not explicitly replace a component with `unspecific` because this might not be permitted in some situations.

```
(define p (->pathname "/usr/blisp/rel15"))
p
     ⇒  #[pathname 71 "/usr/blisp/rel15"]
(pathname-new-name p "rel100")
     ⇒  #[pathname 72 "/usr/blisp/rel100"]
(pathname-new-directory p '(relative "test" "morris"))
     ⇒  #[pathname 73 "test/morris/rel15"]
p
     ⇒  #[pathname 71 "/usr/blisp/rel15"]
```

`pathname-default-device` *pathname device* [procedure]
`pathname-default-directory` *pathname directory* [procedure]
`pathname-default-name` *pathname name* [procedure]
`pathname-default-type` *pathname type* [procedure]
`pathname-default-version` *pathname version* [procedure]
These operations are similar to the `pathname-new-`*`component`* operations, except that they only change the specified *component* if it has the value `#f` in *pathname*.

15.1.3 Operations on Pathnames

`pathname?` *object* [procedure]
Returns `#t` if *object* is a pathname; otherwise returns `#f`.

`pathname=?` *pathname1 pathname2* [procedure]
Returns `#t` if *pathname1* is equivalent to *pathname2*; otherwise returns `#f`. Pathnames are equivalent if all of their components are equivalent, hence two pathnames that are equivalent must identify the same file or equivalent partial pathnames. However, the converse is not true: non-equivalent pathnames may specify the same file (e.g. via absolute and relative directory components), and pathnames that specify no file at all (e.g. name and directory components unspecified) may be equivalent.

`pathname-absolute?` *pathname* [procedure]
Returns `#t` if *pathname* is an absolute rather than relative pathname object; otherwise returns `#f`. Specifically, this procedure returns `#t` when the directory component of *pathname* is a list starting with the symbol `absolute`, and returns `#f` in all other cases. All pathnames are either absolute or relative, so if this procedure returns `#f`, the argument is a relative pathname.

`directory-pathname?` *pathname* [procedure]
Returns `#t` if *pathname* has only directory components and no file components. This is roughly equivalent to

```
(define (directory-pathname? pathname)
  (string-null? (file-namestring pathname)))
```

except that it is faster.

`pathname-wild?` *pathname* [procedure]
Returns `#t` if *pathname* contains any wildcard components; otherwise returns `#f`.

`merge-pathnames` *pathname* [*defaults* [*default-version*]] [procedure]
Returns a pathname whose components are obtained by combining those of *pathname* and *defaults*. *Defaults* defaults to the value of `param:default-pathname-defaults` and *default-version* defaults to `newest`.

The pathnames are combined by components: if *pathname* has a non-missing component, that is the resulting component, otherwise the component from *defaults* is used. The default version can be `#f` to preserve the information that the component was missing from *pathname*. The directory component is handled specially: if both pathnames have directory components that are lists, and the directory component from *pathname* is relative (i.e. starts with `relative`), then the resulting directory component is formed by appending *pathname*'s component to *defaults*'s component. For example:

```
(define path1 (->pathname "scheme/foo.scm"))
(define path2 (->pathname "/usr/morris"))
path1
     ⇒  #[pathname 74 "scheme/foo.scm"]
path2
     ⇒  #[pathname 75 "/usr/morris"]
(merge-pathnames path1 path2)
     ⇒  #[pathname 76 "/usr/scheme/foo.scm"]
(merge-pathnames path2 path1)
     ⇒  #[pathname 77 "/usr/morris.scm"]
```

The merging rules for the version are more complex and depend on whether *pathname* specifies a name. If *pathname* does not specify a name, then the version, if not provided, will come from *defaults*. However, if *pathname* does specify a name then the version is not affected by *defaults*. The reason is that the version "belongs to" some other file name and is unlikely to have anything to do with the new one. Finally, if this process leaves the version missing, then *default-version* is used.

The net effect is that if the user supplies just a name, then the host, device, directory and type will come from *defaults*, but the version will come from *default-version*. If the user supplies nothing, or just a directory, the name, type and version will come over from *defaults* together.

`param:default-pathname-defaults` [parameter]
The value of this parameter (see Section 10.3 [Parameters], page 171) is the default pathname-defaults pathname; if any pathname primitive that needs a set of defaults is not given one, it uses this one. Modifying the `working-directory-pathname` parameter also changes this parameter to the same value, computed by merging the new working directory with the parameter's old value.

`*default-pathname-defaults*` [variable]
This variable is **deprecated**; use `param:default-pathname-defaults` instead.

`pathname-default` *pathname device directory name type version* [procedure]
This procedure defaults all of the components of *pathname* simultaneously. It could have been defined by:

```
(define (pathname-default pathname
                          device directory name type version)
  (make-pathname (pathname-host pathname)
                 (or (pathname-device pathname) device)
                 (or (pathname-directory pathname) directory)
                 (or (pathname-name pathname) name)
                 (or (pathname-type pathname) type)
                 (or (pathname-version pathname) version)))
```

`file-namestring` *pathname* [procedure]
`directory-namestring` *pathname* [procedure]
`host-namestring` *pathname* [procedure]
`enough-namestring` *pathname* [*defaults*] [procedure]
These procedures return a string corresponding to a subset of the *pathname* information. `file-namestring` returns a string representing just the *name*, *type* and *version* components of *pathname*; the result of `directory-namestring` represents just the *host*, *device*, and *directory* components; and `host-namestring` returns a string for just the *host* portion.

`enough-namestring` takes another argument, *defaults*. It returns an abbreviated namestring that is just sufficient to identify the file named by *pathname* when considered relative to the *defaults* (which defaults to the value of `param:default-pathname-defaults`).

```
(file-namestring "/usr/morris/minor.van")
     ⇒  "minor.van"
(directory-namestring "/usr/morris/minor.van")
     ⇒  "/usr/morris/"
(enough-namestring "/usr/morris/men")
     ⇒  "men"       ;perhaps
```

`file-pathname` *pathname* [procedure]
`directory-pathname` *pathname* [procedure]
`enough-pathname` *pathname* [*defaults*] [procedure]
These procedures return a pathname corresponding to a subset of the *pathname* information. `file-pathname` returns a pathname with just the *name*, *type* and *version* components of *pathname*. The result of `directory-pathname` is a pathname containing the *host*, *device* and *directory* components of *pathname*.

`enough-pathname` takes another argument, *defaults*. It returns an abbreviated pathname that is just sufficient to identify the file named by *pathname* when considered relative to the *defaults* (which defaults to the value of `param:default-pathname-defaults`).

These procedures are similar to `file-namestring`, `directory-namestring` and `enough-namestring`, but they return pathnames instead of strings.

`directory-pathname-as-file` *pathname* [procedure]
Returns a pathname that is equivalent to *pathname*, but in which the directory component is represented as a file. The last directory is removed from the directory component and converted into name and type components. This is the inverse operation to `pathname-as-directory`.

```
(directory-pathname-as-file (->pathname "/usr/blisp/"))
     ⇒  #[pathname "/usr/blisp"]
```

`pathname-as-directory` *pathname* [procedure]
Returns a pathname that is equivalent to *pathname*, but in which any file components have been converted to a directory component. If *pathname* does not have name, type, or version components, it is returned without modification. Otherwise, these file components are converted into a string, and the string is added to the end of the list of directory components. This is the inverse operation to `directory-pathname-as-file`.

```
(pathname-as-directory (->pathname "/usr/blisp/rel5"))
     ⇒  #[pathname "/usr/blisp/rel5/"]
```

15.1.4 Miscellaneous Pathname Procedures

This section gives some standard operations on host objects, and some procedures that return some useful pathnames.

`local-host` [variable]
This variable has as its value the host object that describes the local host's file system.

`host?` *object* [procedure]
Returns `#t` if *object* is a pathname host; otherwise returns `#f`.

`host=?` *host1 host2* [procedure]
Returns `#t` if *host1* and *host2* denote the same pathname host; otherwise returns `#f`.

`init-file-pathname` [*host*] [procedure]
Returns a pathname for the user's initialization file on *host*. The *host* argument defaults to the value of `local-host`. If the initialization file does not exist this procedure returns `#f`.

Under unix, the init file is called `.scheme.init`; under Windows, the init file is called `scheme.ini`. In either case, it is located in the user's home directory, which is computed by `user-homedir-pathname`.

`user-homedir-pathname` [*host*] [procedure]
Returns a pathname for the user's "home directory" on *host*. The *host* argument defaults to the value of `local-host`. The concept of a "home directory" is itself somewhat implementation-dependent, but it should be the place where the user keeps personal files, such as initialization files and mail.

Under unix, the user's home directory is specified by the `HOME` environment variable. If this variable is undefined, the user name is computed using the `getlogin` system

call, or if that fails, the `getuid` system call. The resulting user name is passed to the `getpwnam` system call to obtain the home directory.

Under Windows, several heuristics are tried to find the user's home directory. The user's home directory is computed by examining several environment variables, in the following order:

- `HOMEDRIVE` and `HOMEPATH` are both defined and `%HOMEDRIVE%%HOMEPATH%` is an existing directory. (These variables are automatically defined by Windows NT.)
- `HOME` is defined and `%HOME%` is an existing directory.
- `USERDIR` and `USERNAME` are defined and `%USERDIR%\%USERNAME%` is an existing directory.
- `USERDIR` and `USER` are defined and `%USERDIR%\%USER%` is an existing directory.
- `USERNAME` is defined and `%USERNAME%` is an existing directory on the Windows system drive.
- `USER` is defined and `%USER%` is an existing directory on the Windows system drive.
- Finally, if all else fails, the Windows system drive is used as the home directory.

`system-library-pathname` *pathname* [procedure]

Locates *pathname* in MIT/GNU Scheme's system library directory. An error of type `condition-type:file-operation-error` is signalled if *pathname* cannot be located on the library search path.

```
(system-library-pathname "compiler.com")
  ⇒ #[pathname 45 "/usr/local/lib/mit-scheme/compiler.com"]
```

`system-library-directory-pathname` *pathname* [procedure]

Locates the pathname of an MIT/GNU Scheme system library directory. An error of type `condition-type:file-operation-error` is signalled if *pathname* cannot be located on the library search path.

```
(system-library-directory-pathname "options")
     ⇒ #[pathname 44 "/usr/local/lib/mit-scheme/options/"]
```

15.2 Working Directory

When MIT/GNU Scheme is started, the *current working directory* (or simply, *working directory*) is initialized in an operating-system dependent manner; usually, it is the directory in which Scheme was invoked. The working directory can be determined from within Scheme by calling the `pwd` procedure, and changed by calling the `cd` procedure. Each REP loop has its own working directory, and inferior REP loops initialize their working directory from the value in effect in their superior at the time they are created.

`working-directory-pathname` [procedure]
`pwd` [procedure]

Returns the current working directory as a pathname that has no name, type, or version components, just host, device, and directory components. `pwd` is an alias for `working-directory-pathname`; the long name is intended for programs and the short name for interactive use.

`set-working-directory-pathname!` *filename* [procedure]
`cd` *filename* [procedure]

Makes *filename* the current working directory and returns the new current working directory as a pathname. *Filename* is coerced to a pathname using `pathname-as-directory`. `cd` is an alias for `set-working-directory-pathname!`; the long name is intended for programs and the short name for interactive use.

When this procedure is executed in the top-level REP loop, it changes the working directory of the running Scheme executable.

```
(set-working-directory-pathname! "/usr/morris/blisp")
     ⇒  #[pathname "/usr/morris/blisp/"]
(set-working-directory-pathname! "~")
     ⇒  #[pathname "/usr/morris/"]
```

This procedure signals an error if *filename* does not refer to an existing directory.

If *filename* describes a relative rather than absolute pathname, this procedure interprets it as relative to the current working directory, before changing the working directory.

```
(working-directory-pathname)
     ⇒  #[pathname "/usr/morris/"]
(set-working-directory-pathname! "foo")
     ⇒  #[pathname "/usr/morris/foo/"]
```

`with-working-directory-pathname` *filename thunk* [procedure]

This procedure dynamically binds the current working directory to *filename* and returns the value of *thunk* (a procedure of no arguments). *Filename* is coerced to a pathname using `pathname-as-directory`.

15.3 File Manipulation

This section describes procedures that manipulate files and directories. Any of these procedures can signal a number of errors for many reasons. The specifics of these errors are much too operating-system dependent to document here. However, if such an error is signalled by one of these procedures, it will be of type `condition-type:file-operation-error`.

`file-exists?` *filename* [procedure]
`file-exists-direct?` *filename* [procedure]
`file-exists-indirect?` *filename* [procedure]

These procedures return `#t` if *filename* is an existing file or directory; otherwise they return `#f`. In operating systems that support symbolic links, if the file is a symbolic link, `file-exists-direct?` tests for the existence of the link, while `file-exists-indirect?` and `file-exists?` test for the existence of the file pointed to by the link.

`copy-file` *source-filename target-filename* [procedure]

Makes a copy of the file named by *source-filename*. The copy is performed by creating a new file called *target-filename*, and filling it with the same data as *source-filename*.

`rename-file` *source-filename target-filename* [procedure]
Changes the name of *source-filename* to be *target-filename*. In the unix implementation, this will not rename across file systems.

`delete-file` *filename* [procedure]
Deletes the file named *filename*.

`delete-file-no-errors` *filename* [procedure]
Like `delete-file`, but returns a boolean value indicating whether an error occurred during the deletion. If no errors occurred, `#t` is returned. If an error of type `condition-type:file-error` or `condition-type:port-error` is signalled, `#f` is returned.

`hard-link-file` *source-filename target-filename* [procedure]
Makes a hard link from *source-filename* to *target-filename*. This operation gives the file specified by *source-filename* a new name, in addition to the old name.

This currently works only on unix systems. It is further restricted to work only when *source-filename* and *target-filename* refer to names in the same file system.

`soft-link-file` *source-filename target-filename* [procedure]
Creates a new soft link called *target-filename* that points at the file *source-filename*. (Soft links are also sometimes called *symbolic* links.) Note that *source-filename* will be interpreted as a string (although you may specify it as a pathname object, if you wish). The contents of this string will be stored in the file system as the soft link. When a file operation attempts to open the link, the contents of the link are interpreted relative to the link's location at that time.

This currently works only on unix systems.

`make-directory` *filename* [procedure]
Creates a new directory named *filename*. Signals an error if *filename* already exists, or if the directory cannot be created.

`delete-directory` *filename* [procedure]
Deletes the directory named *filename*. Signals an error if the directory does not exist, is not a directory, or contains any files or subdirectories.

`->truename` *filename* [procedure]
This procedure attempts to discover and return the "true name" of the file associated with *filename* within the file system. An error of type `condition-type:file-operation-error` is signalled if the appropriate file cannot be located within the file system.

`call-with-temporary-file-pathname` *procedure* [procedure]
Calls `temporary-file-pathname` to create a temporary file, then calls *procedure* with one argument, the pathname referring to that file. When *procedure* returns, if the temporary file still exists, it is deleted; then, the value yielded by *procedure* is returned. If *procedure* escapes from its continuation, and the file still exists, it is deleted.

`temporary-file-pathname` [*directory*] [procedure]

Creates a new empty temporary file and returns a pathname referring to it. The temporary file is created with Scheme's default permissions, so barring unusual circumstances it can be opened for input and/or output without error. The temporary file will remain in existence until explicitly deleted. If the file still exists when the Scheme process terminates, it will be deleted.

If *directory* is specified, the temporary file will be stored there. If it is not specified, or if it is `#f`, the temporary file will be stored in the directory returned by `temporary-directory-pathname`.

`temporary-directory-pathname` [procedure]

Returns the pathname of an existing directory that can be used to store temporary files. These directory names are tried, in order, until a writeable directory is found:

- The directories specified by the environment variables `TMPDIR`, `TEMP`, or `TMP`.
- Under unix, the directories `/var/tmp`, `/usr/tmp`, or `/tmp`.
- Under Windows, the following directories on the system drive: `\temp`, `\tmp`, or `\`.
- Under Windows, the current directory, as specified by `param:default-pathname-defaults`.

`file-directory?` *filename* [procedure]

Returns `#t` if the file named *filename* exists and is a directory. Otherwise returns `#f`. In operating systems that support symbolic links, if *filename* names a symbolic link, this examines the file linked to, not the link itself.

This is equivalent to

```
(eq? 'directory (file-type-indirect filename))
```

`file-regular?` *filename* [procedure]

Returns `#t` if the file named *filename* exists and is a regular file (i.e. not a directory, symbolic link, device file, etc.). Otherwise returns `#f`. In operating systems that support symbolic links, if *filename* names a symbolic link, this examines the file linked to, not the link itself.

This is equivalent to

```
(eq? 'regular (file-type-indirect filename))
```

`file-symbolic-link?` *filename* [procedure]

In operating systems that support symbolic links, if the file named *filename* exists and is a symbolic link, this procedure returns the contents of the symbolic link as a newly allocated string. The returned value is the name of the file that the symbolic link points to and must be interpreted relative to the directory of *filename*. If *filename* either does not exist or is not a symbolic link, or if the operating system does not support symbolic links, this procedure returns `#f`.

`file-type-direct` *filename* [procedure]
`file-type-indirect` *filename* [procedure]

If the file named *filename* exists, `file-type-direct` returns a symbol specifying what type of file it is. For example, if *filename* refers to a directory, the symbol `directory` is returned. If *filename* doesn't refer to an existing file, `#f` is returned.

If *filename* refers to a symbolic link, `file-type-direct` returns the type of the link itself, while `file-type-indirect` returns the type of the file linked to.

At this time, the symbols that can be returned are the following. The names are intended to be self-explanatory. Most of these names can only be returned on particular operating systems, and so the operating-system name is prefixed to the name.

```
regular
directory
unix-symbolic-link
unix-character-device
unix-block-device
unix-named-pipe
unix-socket
```

`file-readable?` *filename* [procedure]
Returns `#t` if *filename* names a file that can be opened for input; i.e. a *readable* file. Otherwise returns `#f`.

`file-writeable?` *filename* [procedure]
Returns `#t` if *filename* names a file that can be opened for output; i.e. a *writeable* file. Otherwise returns `#f`.

`file-executable?` *filename* [procedure]
Returns `#t` if *filename* names a file that can be executed. Otherwise returns `#f`. Under unix, an executable file is identified by its mode bits. Under Windows, an executable file has one of the file extensions `.exe`, `.com`, or `.bat`.

`file-access` *filename mode* [procedure]
Mode must be an exact integer between `0` and `7` inclusive; it is a bitwise-encoded predicate selector with `1` meaning "executable", `2` meaning "writeable", and `4` meaning "readable". `file-access` returns `#t` if *filename* exists and satisfies the predicates selected by *mode*. For example, if *mode* is `5`, then *filename* must be both readable and executable. If *filename* doesn't exist, or if it does not satisfy the selected predicates, `#f` is returned.

`file-eq?` *filename1 filename2* [procedure]
Determines whether *filename1* and *filename2* refer to the same file. Under unix, this is done by comparing the inodes and devices of the two files. Under Windows, this is done by comparing the filename strings.

`file-modes` *filename* [procedure]
If *filename* names an existing file, `file-modes` returns an exact non-negative integer encoding the file's permissions. The encoding of this integer is operating-system dependent. Under unix, it is the least-significant 12 bits of the `st_mode` element of the `struct stat` structure. Under Windows, it is the file attribute bits, which are described below. If *filename* does not name an existing file, `#f` is returned.

`set-file-modes!` *filename modes* [procedure]
Filename must name an existing file. *Modes* must be an exact non-negative integer that could have been returned by a call to `file-modes`. `set-file-modes!` modifies the file's permissions to be those encoded by *modes*.

`nt-file-mode/read-only` [variable]
`nt-file-mode/hidden` [variable]
`nt-file-mode/system` [variable]
`nt-file-mode/directory` [variable]
`nt-file-mode/archive` [variable]
`nt-file-mode/normal` [variable]
`nt-file-mode/temporary` [variable]
`nt-file-mode/compressed` [variable]

The values of these variables are the "mode bits" that comprise the value returned by `file-modes` under Windows. These bits are small integers that are combined by adding to form a complete set of modes. The integer zero represents a set of modes in which none of these bits are set.

`file-modification-time` *filename* [procedure]

Returns the modification time of *filename* as an exact non-negative integer. The result may be compared to other file times using ordinary integer arithmetic. If *filename* names a file that does not exist, `file-modification-time` returns `#f`.

In operating systems that support symbolic links, if *filename* names a symbolic link, `file-modification-time` returns the modification time of the file linked to. An alternate procedure, `file-modification-time-direct`, returns the modification time of the link itself; in all other respects it is identical to `file-modification-time`. For symmetry, `file-modification-time-indirect` is a synonym of `file-modification-time`.

`file-access-time` *filename* [procedure]

Returns the access time of *filename* as an exact non-negative integer. The result may be compared to other file times using ordinary integer arithmetic. If *filename* names a file that does not exist, `file-access-time` returns `#f`.

In operating systems that support symbolic links, if *filename* names a symbolic link, `file-access-time` returns the access time of the file linked to. An alternate procedure, `file-access-time-direct`, returns the access time of the link itself; in all other respects it is identical to `file-access-time`. For symmetry, `file-access-time-indirect` is a synonym of `file-access-time`.

`set-file-times!` *filename access-time modification-time* [procedure]

Filename must name an existing file, while *access-time* and *modification-time* must be valid file times that might have been returned by `file-access-time` and `file-modification-time`, respectively. `set-file-times!` alters the access and modification times of the file specified by *filename* to the values given by *access-time* and *modification-time*, respectively. For convenience, either of the time arguments may be specified as `#f`; in this case the corresponding time is not changed. `set-file-times!` returns an unspecified value.

`current-file-time` [procedure]

Returns the current time as an exact non-negative integer, in the same format used by the above file-time procedures. This number can be compared to other file times using ordinary arithmetic operations.

`file-touch` *filename* [procedure]
Touches the file named *filename*. If the file already exists, its modification time is set to the current file time and `#f` is returned. Otherwise, the file is created and `#t` is returned. This is an atomic test-and-set operation, so it is useful as a synchronization mechanism.

`file-length` *filename* [procedure]
Returns the length, in bytes, of the file named *filename* as an exact non-negative integer.

`file-attributes` *filename* [procedure]
This procedure determines if the file named *filename* exists, and returns information about it if so; if the file does not exist, it returns `#f`.

In operating systems that support symbolic links, if *filename* names a symbolic link, `file-attributes` returns the attributes of the link itself. An alternate procedure, `file-attributes-indirect`, returns the attributes of the file linked to; in all other respects it is identical to `file-attributes`. For symmetry, `file-attributes-direct` is a synonym of `file-attributes`.

The information returned by `file-attributes` is decoded by accessor procedures. The following accessors are defined in all operating systems:

`file-attributes/type` *attributes* [procedure]
The file type: `#t` if the file is a directory, a character string (the name linked to) if a symbolic link, or `#f` for all other types of file.

`file-attributes/access-time` *attributes* [procedure]
The last access time of the file, an exact non-negative integer.

`file-attributes/modification-time` *attributes* [procedure]
The last modification time of the file, an exact non-negative integer.

`file-attributes/change-time` *attributes* [procedure]
The last change time of the file, an exact non-negative integer.

`file-attributes/length` *attributes* [procedure]
The length of the file in bytes.

`file-attributes/mode-string` *attributes* [procedure]
The mode string of the file, a newly allocated string showing the file's mode bits. Under unix, this string is in unix format. Under Windows, this string shows the standard "DOS" attributes in their usual format.

`file-attributes/n-links` *attributes* [procedure]
The number of links to the file, an exact positive integer. Under Windows, this is always `1`.

The following additional accessors are defined under unix:

`file-attributes/uid` *attributes* [procedure]
The user id of the file's owner, an exact non-negative integer.

`file-attributes/gid` *attributes* [procedure]
The group id of the file's group, an exact non-negative integer.

`file-attributes/inode-number` *attributes* [procedure]
The inode number of the file, an exact non-negative integer.

The following additional accessor is defined under Windows:

`file-attributes/modes` *attributes* [procedure]
The attribute bits of the file. This is an exact non-negative integer containing the file's attribute bits, exactly as specified by the operating system's API.

15.4 Directory Reader

`directory-read` *directory* [*sort?*] [procedure]
Directory must be an object that can be converted into a pathname by `->pathname`. The directory specified by *directory* is read, and the contents of the directory is returned as a newly allocated list of absolute pathnames. The result is sorted according to the usual sorting conventions for directories, unless *sort?* is specified as `#f`. If *directory* has name, type, or version components, the returned list contains only those pathnames whose name, type, and version components match those of *directory*; `wild` or `#f` as one of these components means "match anything".

The Windows implementation supports "globbing", in which the characters `*` and `?` are interpreted to mean "match anything" and "match any character", respectively. This "globbing" is supported only in the file part of *directory*.

15.5 Date and Time

MIT/GNU Scheme provides a simple set of procedures for manipulating date and time information. There are four time representations, each of which serves a different purpose. Each representation may be converted to any of the others.

The primary time representation, *universal time*, is an exact non-negative integer counting the number of seconds that have elapsed since midnight January 1, 1900 UTC. (UTC stands for *Coordinated Universal Time*, and is the modern name for Greenwich Mean Time.) This format is produced by `get-universal-time` and `decoded-time->universal-time`.

The second representation, *decoded time*, is a record structure in which the time is broken down into components, such as month, minute, etc. Decoded time is always relative to a particular time zone, which is a component of the structure. This format is produced by `global-decoded-time` and `local-decoded-time`.

The third representation, *file time*, is an exact non-negative integer that is larger for increasing time. Unlike universal time, this representation is operating-system dependent. This format is produced by all of the file-attribute procedures, for example `file-modification-time` and `file-attributes`.

The fourth representation, the *time string*, is an external representation for time. This format is defined by RFC-822, *Standard for the format of ARPA Internet text messages*, with the modification that years are represented as four-digit numbers rather than two-digit numbers. This format is the standard format for Internet email and numerous other network protocols.

Within this section, argument variables named *universal-time*, *decoded-time*, *file-time*, and *time-string* are respectively required to be of the corresponding format.

15.5.1 Universal Time

`get-universal-time` [procedure]

Return the current time in universal format.

```
(get-universal-time) ⇒ 3131453078
```

`epoch` [variable]

`epoch` is the representation of midnight January 1, 1970 UTC in universal-time format.

```
epoch ⇒ 2208988800
```

15.5.2 Decoded Time

Objects representing standard time components, such as seconds and minutes, are required to be exact non-negative integers. Seconds and minutes must be inclusively between `0` and `59`; hours between `0` and `23`; days between `1` and `31`; months between `1` and `12`; years are represented in "four-digit" form, in which 1999 is represented as `1999` — *not* `99`.

`local-decoded-time` [procedure]

Return the current time in decoded format. The decoded time is represented in the local time zone.

```
(pp (local-decoded-time))
⊣ #[decoded-time 76]
⊣ (second 2)
⊣ (minute 12)
⊣ (hour 11)
⊣ (day 27)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 1)
⊣ (daylight-savings-time 1)
⊣ (zone 5)
```

`global-decoded-time` [procedure]

Return the current time in decoded format. The decoded time is represented in UTC.

```
(pp (global-decoded-time))
⊣ #[decoded-time 77]
⊣ (second 8)
⊣ (minute 12)
⊣ (hour 15)
⊣ (day 27)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 1)
⊣ (daylight-savings-time 0)
⊣ (zone 0)
```

`make-decoded-time` *second minute hour day month year* [*zone*] [procedure]
Return a new decoded-time object representing the given time. The arguments must be valid components according to the above rules, and must form a valid date.

If *zone* is not supplied or is `#f`, the resulting decoded time will be represented in the local time zone. Otherwise, *zone* must be a valid time zone, and the result will be represented in that zone.

Warning: because this procedure depends on the operating system's runtime library, it is not capable of representing all dates. In particular, on most unix systems, it is not possible to encode dates that occur prior to midnight, January 1, 1970 UTC. Attempting to do this will signal an error.

```
(pp (make-decoded-time 0 9 11 26 3 1999))
⊣ #[decoded-time 19]
⊣ (second 0)
⊣ (minute 9)
⊣ (hour 11)
⊣ (day 26)
⊣ (month 3)
⊣ (year 1999)
⊣ (day-of-week 4)
⊣ (daylight-savings-time 0)
⊣ (zone 5)

(pp (make-decoded-time 0 9 11 26 3 1999 3))
⊣ #[decoded-time 80]
⊣ (second 0)
⊣ (minute 9)
⊣ (hour 11)
⊣ (day 26)
⊣ (month 3)
⊣ (year 1999)
⊣ (day-of-week 4)
⊣ (daylight-savings-time 0)
⊣ (zone 3)
```

`decoded-time/second` *decoded-time* [procedure]
`decoded-time/minute` *decoded-time* [procedure]
`decoded-time/hour` *decoded-time* [procedure]
`decoded-time/day` *decoded-time* [procedure]
`decoded-time/month` *decoded-time* [procedure]
`decoded-time/year` *decoded-time* [procedure]
Return the corresponding component of *decoded-time*.

```
(decoded-time/second (local-decoded-time)) ⇒ 17
(decoded-time/year (local-decoded-time)) ⇒ 1999
(decoded-time/day (local-decoded-time)) ⇒ 26
```

`decoded-time/day-of-week` *decoded-time* [procedure]
Return the day of the week on which *decoded-time* falls, encoded as an exact integer between `0` (Monday) and `6` (Sunday), inclusive.

```
(decoded-time/day-of-week (local-decoded-time)) ⇒ 4
```

`decoded-time/daylight-savings-time?` *decoded-time* [procedure]
Return `#t` if *decoded-time* is represented using daylight savings time. Otherwise return `#f`.

```
(decoded-time/daylight-savings-time? (local-decoded-time))
                  ⇒ #f
```

`decoded-time/zone` *decoded-time* [procedure]
Return the time zone in which *decoded-time* is represented. This is an exact rational number between `-24` and `+24` inclusive, that when multiplied by `3600` is an integer. The value is the number of hours west of UTC.

```
(decoded-time/zone (local-decoded-time)) ⇒ 5
```

`time-zone?` *object* [procedure]
Returns `#t` if *object* is an exact number between `-24` and `+24` inclusive, that when multiplied by `3600` is an integer.

```
(time-zone? -5)   ⇒ #t
(time-zone? 11/2) ⇒ #t
(time-zone? 11/7) ⇒ #f
```

`month/max-days` *month* [procedure]
Returns the maximum number of days possible in *month*. *Month* must be an exact integer between `1` and `12` inclusive.

```
(month/max-days 2) ⇒ 29
(month/max-days 3) ⇒ 31
(month/max-days 4) ⇒ 30
```

15.5.3 File Time

As stated above, file time is operating-system dependent. As of this writing, two formats are used. For unix and Windows systems, file time is the number of seconds since midnight January 1, 1970 UTC (the standard unix time convention).

The following procedures generate their results in file-time format:

```
file-access-time
file-access-time-direct
file-access-time-indirect
file-modification-time
file-modification-time-direct
file-modification-time-indirect
file-attributes/access-time
file-attributes/modification-time
file-attributes/change-time
```

Additionally, `set-file-times!` accepts its time arguments in file-time format.

15.5.4 Time-Format Conversion

The procedures described in this section convert times from one format to another.

`universal-time->local-decoded-time` *universal-time* [procedure]
`universal-time->global-decoded-time` *universal-time* [procedure]
Converts an argument in universal-time format to decoded-time format. The result is in the local time zone or UTC, respectively.

```
(pp (universal-time->local-decoded-time (get-universal-time)))
⊣ #[decoded-time 21]
⊣ (second 23)
⊣ (minute 57)
⊣ (hour 17)
⊣ (day 29)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 3)
⊣ (daylight-savings-time 1)
⊣ (zone 5)

(pp (universal-time->global-decoded-time
     (get-universal-time)))
⊣ #[decoded-time 22]
⊣ (second 27)
⊣ (minute 57)
⊣ (hour 21)
⊣ (day 29)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 3)
⊣ (daylight-savings-time 0)
⊣ (zone 0)
```

`universal-time->file-time` *universal-time* [procedure]
Converts an argument in universal-time format to file-time format.

```
(universal-time->file-time (get-universal-time))
    ⇒ 925422988
```

`universal-time->local-time-string` *universal-time* [procedure]
`universal-time->global-time-string` *universal-time* [procedure]
Converts an argument in universal-time format to a time string. The result is in the local time zone or UTC, respectively.

```
(universal-time->local-time-string (get-universal-time))
    ⇒ "Thu, 29 Apr 1999 17:55:31 -0400"
(universal-time->global-time-string (get-universal-time))
    ⇒ "Thu, 29 Apr 1999 21:55:51 +0000"
```

`decoded-time->universal-time` *decoded-time* [procedure]
Converts an argument in decoded-time format to universal-time format.

```
(decoded-time->universal-time (local-decoded-time))
    ⇒ 3134411942
(decoded-time->universal-time (global-decoded-time))
    ⇒ 3134411947
```

`decoded-time->file-time` *decoded-time* [procedure]

Converts an argument in decoded-time format to file-time format.

```
(decoded-time->file-time (local-decoded-time))
    ⇒ 925423191
(decoded-time->file-time (global-decoded-time))
    ⇒ 925423195
```

`decoded-time->string` *decoded-time* [procedure]

Convert an argument in decoded-time format to a time string.

```
(decoded-time->string (local-decoded-time))
    ⇒ "Thu, 29 Apr 1999 18:00:43 -0400"
(decoded-time->string (global-decoded-time))
    ⇒ "Thu, 29 Apr 1999 22:00:46 +0000"
```

`file-time->universal-time` *file-time* [procedure]

Converts an argument in universal-time format to file-time format.

```
(file-time->universal-time (file-modification-time "/"))
    ⇒ 3133891907
```

`file-time->local-decoded-time` *file-time* [procedure]
`file-time->global-decoded-time` *file-time* [procedure]

Converts an argument in file-time format to decoded-time format. The result is in the local time zone or UTC, respectively.

```
(pp (file-time->local-decoded-time
     (file-modification-time "/")))
⊣ #[decoded-time 26]
⊣ (second 47)
⊣ (minute 31)
⊣ (hour 17)
⊣ (day 23)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 4)
⊣ (daylight-savings-time 1)
⊣ (zone 5)
```

```
(pp (file-time->global-decoded-time
     (file-modification-time "/")))
⊣ #[decoded-time 27]
⊣ (second 47)
⊣ (minute 31)
⊣ (hour 21)
⊣ (day 23)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 4)
⊣ (daylight-savings-time 0)
⊣ (zone 0)
```

`file-time->local-time-string` *file-time* [procedure]
`file-time->global-time-string` *file-time* [procedure]

Converts an argument in file-time format to a time string. The result is in the local time zone or UTC, respectively.

```
(file-time->local-time-string (file-modification-time "/"))
    ⇒ "Fri, 23 Apr 1999 17:31:47 -0400"
(file-time->global-time-string (file-modification-time "/"))
    ⇒ "Fri, 23 Apr 1999 21:31:47 +0000"
```

`string->universal-time` *time-string* [procedure]

Converts a time-string argument to universal-time format.

```
(string->universal-time "Fri, 23 Apr 1999 21:31:47 +0000")
    ⇒ 3133888307
(string->universal-time "Fri, 23 Apr 1999 17:31:47 -0400")
    ⇒ 3133888307
```

`string->decoded-time` *time-string* [procedure]

Converts a time-string argument to decoded-time format.

```
(pp (string->decoded-time "Fri, 23 Apr 1999 17:31:47 -0400"))
⊣ #[decoded-time 30]
⊣ (second 47)
⊣ (minute 31)
⊣ (hour 17)
⊣ (day 23)
⊣ (month 4)
⊣ (year 1999)
⊣ (day-of-week 4)
⊣ (daylight-savings-time 0)
⊣ (zone 4)
```

`string->file-time` *time-string* [procedure]

Converts a time-string argument to file-time format.

```
(string->file-time "Fri, 23 Apr 1999 17:31:47 -0400")
    ⇒ 924899507
```

15.5.5 External Representation of Time

The normal external representation for time is the time string, as described above. The procedures in this section generate alternate external representations of time which are more verbose and may be more suitable for presentation to human readers.

`decoded-time/date-string` *decoded-time* [procedure]
`decoded-time/time-string` *decoded-time* [procedure]
These procedures return strings containing external representations of the date and time, respectively, represented by *decoded-time*. The results are implicitly in local time.

```
(decoded-time/date-string (local-decoded-time))
    ⇒ "Tuesday March 30, 1999"
(decoded-time/time-string (local-decoded-time))
    ⇒ "11:22:38 AM"
```

`day-of-week/long-string` *day-of-week* [procedure]
`day-of-week/short-string` *day-of-week* [procedure]
Returns a string representing the given *day-of-week*. The argument must be an exact non-negative integer between `0` and `6` inclusive. `day-of-week/long-string` returns a long string that fully spells out the name of the day. `day-of-week/short-string` returns a shortened string that abbreviates the day to three letters.

```
(day-of-week/long-string 0)  ⇒ "Monday"
(day-of-week/short-string 0) ⇒ "Mon"
(day-of-week/short-string 3) ⇒ "Thu"
```

`month/long-string` *month* [procedure]
`month/short-string` *month* [procedure]
Returns a string representing the given *month*. The argument must be an exact non-negative integer between `1` and `12` inclusive. `month/long-string` returns a long string that fully spells out the name of the month. `month/short-string` returns a shortened string that abbreviates the month to three letters.

```
(month/long-string 1)   ⇒ "January"
(month/short-string 1)  ⇒ "Jan"
(month/short-string 10) ⇒ "Oct"
```

`time-zone->string` [procedure]
Returns a string corresponding to the given time zone. This string is the same string that is used to generate RFC-822 time strings.

```
(time-zone->string 5)    ⇒ "-0500"
(time-zone->string -4)   ⇒ "+0400"
(time-zone->string 11/2) ⇒ "-0530"
```

15.6 Machine Time

The previous section dealt with procedures that manipulate clock time. This section describes procedures that deal with computer time: elapsed CPU time, elapsed real time, and so forth. These procedures are useful for measuring the amount of time it takes to execute code.

Some of the procedures in this section manipulate a time representation called *ticks*. A tick is a unit of time that is unspecified here but can be converted to and from seconds by supplied procedures. A count in ticks is represented as an exact integer. At present each tick is one millisecond, but this may change in the future.

`process-time-clock` [procedure]
Returns the amount of process time, in ticks, that has elapsed since Scheme was started. Process time is measured by the operating system and is time during which the Scheme process is computing. It does not include time in system calls, but depending on the operating system it may include time used by subprocesses.

```
(process-time-clock) ⇒ 21290
```

`real-time-clock` [procedure]
Returns the amount of real time, in ticks, that has elapsed since Scheme was started. Real time is the time measured by an ordinary clock.

```
(real-time-clock) ⇒ 33474836
```

`internal-time/ticks->seconds` *ticks* [procedure]
Returns the number of seconds corresponding to *ticks*. The result is always a real number.

```
(internal-time/ticks->seconds 21290) ⇒ 21.29
(internal-time/ticks->seconds 33474836) ⇒ 33474.836
```

`internal-time/seconds->ticks` *seconds* [procedure]
Returns the number of ticks corresponding to *seconds*. *Seconds* must be a real number.

```
(internal-time/seconds->ticks 20.88) ⇒ 20880
(internal-time/seconds->ticks 20.83) ⇒ 20830
```

`system-clock` [procedure]
Returns the amount of process time, in seconds, that has elapsed since Scheme was started. Roughly equivalent to:

```
(internal-time/ticks->seconds (process-time-clock))
```

Example:

```
(system-clock) ⇒ 20.88
```

`runtime` [procedure]
Returns the amount of process time, in seconds, that has elapsed since Scheme was started. However, it does not include time spent in garbage collection.

```
(runtime) ⇒ 20.83
```

`with-timings` *thunk receiver* [procedure]
Calls *thunk* with no arguments. After *thunk* returns, *receiver* is called with three arguments describing the time spent while computing *thunk*: the elapsed run time, the amount of time spent in the garbage collector, and the elapsed real time. All three times are in ticks.

This procedure is most useful for doing performance measurements, and is designed to have relatively low overhead.

```
(with-timings
 (lambda () ... hairy computation ...)
 (lambda (run-time gc-time real-time)
   (write (internal-time/ticks->seconds run-time))
   (write-char #\space)
   (write (internal-time/ticks->seconds gc-time))
   (write-char #\space)
   (write (internal-time/ticks->seconds real-time))
   (newline)))
```

`measure-interval` *runtime? procedure* [procedure]

Calls *procedure*, passing it the current process time, in seconds, as an argument. The result of this call must be another procedure. When *procedure* returns, the resulting procedure is tail-recursively called with the ending time, in seconds, as an argument.

If *runtime?* is `#f`, the elapsed time is deducted from the elapsed system time returned by `runtime`.

While this procedure can be used for time measurement, its interface is somewhat clumsy for that purpose. We recommend that you use `with-timings` instead, because it is more convenient and has lower overhead.

```
(measure-interval #t
                  (lambda (start-time)
                    (let ((v ... hairy computation ...))
                      (lambda (end-time)
                        (write (- end-time start-time))
                        (newline)
                        v))))
```

15.7 Subprocesses

MIT/GNU Scheme provides the ability to run and control subprocesses. This support is divided into two parts: a low-level set of primitives that maps onto the underlying operating system's process-control primitives, and a high-level set of procedures for starting a subprocess and running it to completion in a single call. Subprocesses that are run in the latter fashion are referred to as *synchronous*, because they are started and stopped in synchrony with a Scheme procedure call.

This chapter documents Scheme's high-level synchronous-subprocess support. The low-level support is not documented but is available for those who are willing to read the source code.

Synchronous-subprocess support is a run-time-loadable option. To use it, execute

```
(load-option 'synchronous-subprocess)
```

once before calling it.

15.7.1 Subprocess Procedures

There are two commands for running synchronous subprocesses under Scheme. `run-shell-command` is very simple to use, provides access to all shell features, and is to be preferred in most situations. `run-synchronous-subprocess` allows direct execution of a program and precise control of the command-line arguments passed to the program, but does not provide file globbing, I/O redirection, or other shell features.

`run-shell-command` *command option . . .* [procedure]
Runs *command*, which must be a string. *Command* is passed to a command shell for interpretation; how the shell is chosen is detailed below.

The *option*s are a sequence of keyword/value pairs that specify optional behavior. See Section 15.7.3 [Subprocess Options], page 302.

`run-shell-command` waits until the subprocess completes its execution and returns the exit code from the subprocess. If the subprocess is killed or stopped, an error is signalled and the procedure does not return.

`run-synchronous-subprocess` *program arguments option . . .* [procedure]
Runs *program*, passing it the given command-line *arguments*. *Program* must be either the name of a program on the path, or else a pathname to a specific program. *Arguments* must be a list of strings; each string is a single command-line argument to the program.

The *option*s are a sequence of keyword/value pairs that specify optional behavior. See Section 15.7.3 [Subprocess Options], page 302.

`run-synchronous-subprocess` waits until the subprocess completes its execution and returns the exit code from the subprocess. If the subprocess is killed or stopped, an error is signalled and the procedure does not return.

15.7.2 Subprocess Conditions

If a subprocess spawned by one of the above procedures is killed or suspended, then one of the following errors will be signalled.

`condition-type:subprocess-signalled` *subprocess reason* [condition type]
This condition type is a subtype of `condition-type:subprocess-abnormal-termination`. It is signalled when the subprocess is killed.

Subprocess is an object that represents the subprocess involved. The internals of this object can be accessed but the interface is not documented at this time; see the source code for details.

Reason is interesting only on unix systems, where it is the signal that killed the process. On other systems it has a fixed value that conveys no useful information.

`condition-type:subprocess-stopped` *subprocess reason* [condition type]
This condition type is a subtype of `condition-type:subprocess-abnormal-termination`. It is signalled when the subprocess is stopped or suspended.

Subprocess is an object that represents the subprocess involved. The internals of this object can be accessed but the interface is not documented at this time; see the source code for details.

Reason is interesting only on unix systems, where it is the signal that stopped the process. On other systems it has a fixed value that conveys no useful information.

`condition-type:subprocess-abnormal-termination` [condition type]
subprocess reason
This condition type is a subtype of `condition-type:error`. This is an abstract type that is never signalled. It is provided so that condition handlers can be bound to it.

15.7.3 Subprocess Options

The following subprocess options may be passed to `run-shell-command` or `run-synchronous-subprocess`. These options are passed as alternating keyword/value pairs, for example:

```
(run-shell-command "ls /"
                   'output my-output-port
                   'output-buffer-size 8192)
```

The example shows a shell command being run with two options specified: `output` and `output-buffer-size`.

`input` *port* [subprocess option]
Specifies the standard input of the subprocess. *Port* may be an input port, in which case characters are read from *port* and fed to the subprocess until *port* reaches end-of-file. Alternatively, *port* may be `#f`, indicating that the subprocess has no standard input.

The default value of this option is `#f`.

```
(call-with-input-file "foo.in"
  (lambda (port)
    (run-shell-command "cat > /dev/null" 'input port)))
```

`input-line-translation` *line-ending* [subprocess option]
Specifies how line-endings should be translated when writing characters to the subprocess. Ignored if the `input` option is `#f`. *Line-ending* must be either a string specifying the line ending, or the symbol `default`, meaning to use the operating system's standard line ending. In either case, newline characters to be written to the `input` port are translated to the specified line ending before being written.

The default value of this option is `default`.

```
(call-with-input-file "foo.in"
  (lambda (port)
    (run-shell-command "cat > /dev/null"
                       'input port
                       'input-line-translation "\r\n")))
```

`input-buffer-size` *n* [subprocess option]
Specifies the size of the input buffer for the standard input of the subprocess. (This is the buffer on the Scheme side, and has nothing to do with any buffering done on the subprocess side.) Ignored if the `input` option is `#f`. *N* must be an exact positive integer specifying the number of characters the buffer can hold.

The default value of this option is `512`.

```
(call-with-input-file "foo.in"
  (lambda (port)
    (run-shell-command "cat > /dev/null"
                       'input port
                       'input-buffer-size 4096)))
```

`output` *port* [subprocess option]

Specifies the standard output and standard error of the subprocess. *Port* may be an output port, in which case characters are read from the subprocess and fed to *port* until the subprocess finishes. Alternatively, *port* may be `#f`, indicating that the subprocess has no standard output or standard error.

The default value of this option is the value of `(current-output-port)`.

```
(call-with-output-file "foo.out"
  (lambda (port)
    (run-shell-command "ls -la /etc" 'output port)))
```

`output-line-translation` *line-ending* [subprocess option]

Specifies how line-endings should be translated when reading characters from the standard output of the subprocess. Ignored if the `output` option is `#f`. *Line-ending* must be either a string specifying the line ending, or the symbol `default`, meaning to use the operating system's standard line ending. In either case, newline characters read from the subprocess port are translated to the specified line ending.

The default value of this option is `default`.

```
(call-with-output-file "foo.out"
  (lambda (port)
    (run-shell-command "ls -la /etc"
                       'output port
                       'output-line-translation "\r\n")))
```

`output-buffer-size` *n* [subprocess option]

Specifies the size of the output buffer for the standard output of the subprocess. (This is the buffer on the Scheme side, and has nothing to do with any buffering done on the subprocess side.) Ignored if the `output` option is `#f`. *N* must be an exact positive integer specifying the number of characters the buffer can hold.

The default value of this option is `512`.

```
(call-with-output-file "foo.out"
  (lambda (port)
    (run-shell-command "ls -la /etc"
                       'output port
                       'output-buffer-size 4096)))
```

`redisplay-hook` *thunk* [subprocess option]

Specifies that *thunk* is to be run periodically when output from the subprocess is available. *Thunk* must be a procedure of no arguments, or `#f` indicating that no hook is supplied. This option is mostly useful for interactive systems. For example, the Edwin text editor uses this to update output buffers when running some subprocesses.

The default value of this option is `#f`.

```
(run-shell-command "ls -la /etc"
                   'redisplay-hook
                   (lambda ()
                     (update-buffer-contents buffer)))
```

`environment` *environment* [subprocess option]

Specifies the environment variables that are to be used for the subprocess. *Environment* must be either a vector of strings or `#f` indicating the default environment. If it is a vector of strings, each string must be a name/value pair where the name and value are separated by an equal sign, for example, `"foo=bar"`. To define a variable with no value, just omit the value, as in `"foo="`.

Note that the variable `scheme-subprocess-environment` is bound to the default subprocess environment.

The default value of this option is `#f`.

```
(run-shell-command "ls -la /etc"
                   'environment
                   (let* ((v scheme-subprocess-environment)
                          (n (vector-length v))
                          (v (vector-grow v (+ n 1))))
                     (vector-set! v n "TERM=none")
                     v))
```

`working-directory` *pathname* [subprocess option]

Specifies the working directory in which the subprocess will run.

The default value of this option is `(working-directory-pathname)`.

```
(run-shell-command "ls -la" 'working-directory "/etc/")
```

`use-pty?` *boolean* [subprocess option]

This option is meaningful only on unix systems; on other systems it is ignored. Specifies whether to communicate with the subprocess using PTY devices; if true, PTYs will be used, otherwise pipes will be used.

The default value of this option is `#f`.

```
(run-shell-command "ls -la /etc" 'use-pty? #t)
```

`shell-file-name` *pathname* [subprocess option]

Specifies the shell program to use for `run-shell-command`.

The default value of this option is `(os/shell-file-name)`. This is the value of the environment variable `SHELL`, or if `SHELL` is not set, the value is operating-system dependent as follows:

- On unix systems, `/bin/sh` is used.
- On Windows systems, the value of the environment variable `COMSPEC` is used. If that is not set, `cmd.exe` is used for Windows NT, or `command.com` is used for Windows 9x; in each case the shell is found by searching the path.

```
(run-shell-command "ls -la /etc"
                   'shell-file-name "/usr/local/bin/bash")
```

15.8 TCP Sockets

MIT/GNU Scheme provides access to *sockets*, which are a mechanism for inter-process communication. TCP stream sockets are supported, which communicate between computers over a TCP/IP network. TCP sockets are supported on all operating systems.

TCP sockets have two distinct interfaces: one interface to implement a *client* and another to implement a *server*. The basic protocol is that servers set up a listening port and wait for connections from clients. Implementation of clients is simpler and will be treated first.

The socket procedures accept two special arguments, called *host-name* and *service*. *Host-name* is a string which must be the name of an internet host. It is looked up using the ordinary lookup rules for your computer. For example, if your host is `foo.mit.edu` and *host-name* is `"bar"`, then it specifies `bar.mit.edu`.

Service specifies the service to which you will connect. A networked computer normally provides several different services, such as telnet or FTP. Each service is associated with a unique *port number*; for example, the `"www"` service is associated with port `80`. The *service* argument specifies the port number, either as a string, or directly as an exact non-negative integer. Port strings are decoded by the operating system using a table; for example, on unix the table is in `/etc/services`. Usually you will use a port string rather than a number.

`open-tcp-stream-socket` *host-name service* [procedure]

`open-tcp-stream-socket` opens a connection to the host specified by *host-name*. *Host-name* is looked up using the ordinary lookup rules for your computer. The connection is established to the service specified by *service*. The returned value is an I/O port, to which you can read and write characters using ordinary Scheme I/O procedures such as `read-char` and `write-char`.

When you wish to close the connection, just use `close-port`.

As an example, here is how you can open a connection to a web server:

```
(open-tcp-stream-socket "web.mit.edu" "www")
```

Next we will treat setting up a TCP server, which is slightly more complicated. Creating a server is a two-part process. First, you must open a *server socket*, which causes the operating system to listen to the network on a port that you specify. Once the server socket is opened, the operating system will allow clients to connect to your computer on that port.

In the second step of the process, you *accept* the connection, which completes the connection initiated by the client, and allows you to communicate with the client. Accepting a connection does not affect the server socket; it continues to listen for additional client connections. You can have multiple client connections to the same server socket open simultaneously.

`open-tcp-server-socket` *service* [*address*] [procedure]

This procedure opens a server socket that listens for connections to *service*; the socket will continue to listen until you close it. The returned value is a server socket object.

An error is signalled if another process is already listening on the service. Additionally, ports whose number is less than `1024` are privileged on many operating systems, and cannot be used by non-privileged processes; if *service* specifies such a port and you do not have administrative privileges, an error may be signalled.

The optional argument *address* specifies the IP address on which the socket will listen. If this argument is not supplied or is given as `#f`, then the socket listens on all IP addresses for this machine. (This is equivalent to passing the result of calling `host-address-any`.)

`tcp-server-connection-accept` *server-socket block? peer-address* [*line-translation*] [procedure]

Checks to see if a client has connected to *server-socket*. If so, an I/O port is returned. The returned port can be read and written using ordinary Scheme I/O procedures such as `read-char` and `write-char`.

The argument *block?* says what to do if no client has connected at the time of the call. If `#f`, it says to return immediately with two values of `#f`. Otherwise, the call waits until a client connects.

The argument *peer-address* is either `#f` or an IP address as allocated by `allocate-host-address`. If it is an IP address, the address is modified to be the address of the client making the connection.

The optional argument *line-translation* specifies how end-of-line characters will be translated when reading or writing to the returned socket. If this is unspecified or `#f`, then lines will be terminated by CR-LF, which is the standard for most internet protocols. Otherwise, it must be a string, which specifies the line-ending character sequence to use.

Note that closing the port returned by this procedure does not affect *server-socket*; it just closes the particular client connection that was opened by the call. To close *server-socket*, use `close-tcp-server-socket`.

`close-tcp-server-socket` *server-socket* [procedure]

Closes the server socket *server-socket*. The operating system will cease listening for network connections to that service. Client connections to *server-socket* that have already been accepted will not be affected.

15.9 Miscellaneous OS Facilities

This section contains assorted operating-system facilities that don't fit into other categories.

`microcode-id/operating-system` [variable]
`microcode-id/operating-system-name` [obsolete variable]

`microcode-id/operating-system` is bound to a symbol that specifies the type of operating system that Scheme is running under. There are two possible values: `unix` or `nt`.

The **deprecated** variable `microcode-id/operating-system-name` is a string that's equivalent to `microcode-id/operating-system`.

`microcode-id/operating-system-variant` [variable]

This variable is a string that identifies the particular variant of the operating system that Scheme is running under. Here are some of the possible values:

```
"GNU/Linux"
"MacOSX"
"Microsoft Windows NT 4.0 (Build 1381; Service Pack 3)"
```

For Windows systems, it is recommended that you match on the prefix of this string and ignore the `"Build"` suffix. This is because the suffix may contain information about service packs or fixes, while the prefix will be constant for a particular version of Windows.

The next few procedures provide access to the *domain name service* (DNS), which maintains associations between internet host names such as `"www.swiss.ai.mit.edu"` and IP addresses, such as `18.23.0.16`. In MIT/GNU Scheme, we represent an internet host name as a string, and an IP address as a byte vector of length 4 (byte vectors are just character strings that are accessed using `vector-8b-ref` rather than `string-ref`). The bytes in an IP address read in the same order as they do when written out:

```
(get-host-by-name "www.swiss") ⇒ #("\022\027\000\020")
```

`get-host-by-name` *host-name* [procedure]

Looks up the internet host name *host-name* using the DNS, returning a vector of IP addresses for the corresponding host, or `#f` if there is no such host. Usually the returned vector has only one element, but if a host has more than one network interface, the vector might have more than one element.

```
(get-host-by-name "www.swiss") ⇒ #("\022\027\000\020")
```

`get-host-by-address` *ip-address* [procedure]

Does a reverse DNS lookup on *ip-address*, returning the internet host name corresponding to that address, or `#f` if there is no such host.

```
(get-host-by-address "\022\027\000\020") ⇒ "swissnet.ai.mit.edu"
```

`canonical-host-name` *host-name* [procedure]

Finds the "canonical" internet host name for *host-name*. For example:

```
(canonical-host-name "zurich")    ⇒ "zurich.ai.mit.edu"
(canonical-host-name "www.swiss") ⇒ "swissnet.ai.mit.edu"
```

In both examples, the default internet domain '`ai.mit.edu`' is added to *host-name*. In the second example, `"www.swiss"` is an alias for another computer named `"swissnet"`.

`get-host-name` [procedure]

Returns the string that identifies the computer that MIT/GNU Scheme is running on. Usually this is an unqualified internet host name, i.e. the host name without the domain suffix:

```
(get-host-name) ⇒ "aarau"
```

`os/hostname` [procedure]

Returns the canonical internet host name of the computer that MIT/GNU Scheme is running on. So, in contrast to the example for `get-host-name`:

```
(os/hostname) ⇒ "aarau.ai.mit.edu"
```

`allocate-host-address` [procedure]

Allocates and returns an IP address object. This is just a string of a fixed length (current 4 bytes) into which an IP address may be stored. This procedure is used to generate an appropriate argument to be passed to `tcp-server-connection-accept`.

```
(allocate-host-address) ⇒ "Xe\034\241"
```

`host-address-any` [procedure]

Return an IP address object that specifies "any host". This object is useful only when passed as the *address* argument to `open-tcp-server-socket`.

```
(host-address-any) ⇒ "\000\000\000\000"
```

`host-address-loopback` [procedure]

Return an IP address object that specifies the local *loopback* network interface. The loopback interface is a software network interface that can be used only for communicating between processes on the same computer. This address object is useful only when passed as the *address* argument to `open-tcp-server-socket`.

```
(host-address-loopback) ⇒ "\177\000\000\001"
```

16 Error System

The MIT/GNU Scheme error system provides a uniform mechanism for the signalling of errors and other exceptional conditions. The simplest and most generally useful procedures in the error system are:

`error` is used to signal simple errors, specifying a message and some irritant objects (see Section 16.1 [Condition Signalling], page 310). Errors are usually handled by stopping the computation and putting the user in an error REPL.

`warn` is used to signal warnings (see Section 16.1 [Condition Signalling], page 310). Warnings are usually handled by printing a message on the console and continuing the computation normally.

`ignore-errors`
is used to suppress the normal handling of errors within a given dynamic extent (see Section 16.3 [Condition Handling], page 313). Any error that occurs within the extent is trapped, returning immediately to the caller of `ignore-errors`.

More demanding applications require more powerful facilities. To give a concrete example, suppose you want floating-point division to return a very large number whenever the denominator is zero. This behavior can be implemented using the error system.

The Scheme arithmetic system can signal many different kinds of errors, including floating-point divide by zero. In our example, we would like to handle this particular condition specially, allowing the system to handle other arithmetic errors in its usual way.

The error system supports this kind of application by providing mechanisms for distinguishing different types of error conditions and for specifying where control should be transferred should a given condition arise. In this example, there is a specific object that represents the "floating-point divide by zero" condition type, and it is possible to dynamically specify an arbitrary Scheme procedure to be executed when a condition of that type is signalled. This procedure then finds the stack frame containing the call to the division operator, and returns the appropriate value from that frame.

Another useful kind of behavior is the ability to specify uniform handling for related classes of conditions. For example, it might be desirable, when opening a file for input, to gracefully handle a variety of different conditions associated with the file system. One such condition might be that the file does not exist, in which case the program will try some other action, perhaps opening a different file instead. Another related condition is that the file exists, but is read protected, so it cannot be opened for input. If these or any other related conditions occur, the program would like to skip this operation and move on to something else.

At the same time, errors unrelated to the file system should be treated in their usual way. For example, calling `car` on the argument `3` should signal an error. Or perhaps the name given for the file is syntactically incorrect, a condition that probably wants to be handled differently from the case of the file not existing.

To facilitate the handling of classes of conditions, the error system taxonomically organizes all condition types. The types are related to one another by *taxonomical links*, which specify that one type is a "kind of" another type. If two types are linked this way, one is considered to be a *specialization* of the other; or vice-versa, the second is a *generalization*

of the first. In our example, all of the errors associated with opening an input file would be specializations of the condition type "cannot open input file".

The taxonomy of condition types permits any condition type to have no more than one immediate generalization. Thus, the condition types form a forest (set of trees). While users can create new trees, the standard taxonomy (see Section 16.7 [Taxonomy], page 324) is rooted at `condition-type:serious-condition`, `condition-type:warning`, `condition-type:simple-condition`, and `condition-type:breakpoint`; users are encouraged to add new subtypes to these condition types rather than create new trees in the forest.

To summarize, the error system provides facilities for the following tasks. The sections that follow will describe these facilities in more detail.

Signalling a condition
: A condition may be signalled in a number of different ways. Simple errors may be signalled, without explicitly defining a condition type, using `error`. The `signal-condition` procedure provides the most general signalling mechanism.

Handling a condition
: The programmer can dynamically specify handlers for particular condition types or for classes of condition types, by means of the `bind-condition-handler` procedure. Individual handlers have complete control over the handling of a condition, and additionally may decide not to handle a particular condition, passing it on to previously bound handlers.

Restarting from a handler
: The `with-restart` procedure provides a means for condition-signalling code to communicate to condition-handling code what must be done to proceed past the condition. Handlers can examine the restarts in effect when a condition was signalled, allowing a structured way to continue an interrupted computation.

Packaging condition state
: Each condition is represented by an explicit object. Condition objects contain information about the nature of the condition, information that describes the state of the computation from which the condition arose, and information about the ways the computation can be restarted.

Classification of conditions
: Each condition has a type, represented by a condition type object. Each condition type may be a specialization of some other condition types. A group of types that share a common generalization can be handled uniformly by specifying a handler for the generalization.

16.1 Condition Signalling

Once a condition instance has been created using `make-condition` (or any condition constructor), it can be *signalled*. The act of signalling a condition is separated from the act of creating the condition to allow more flexibility in how conditions are handled. For example, a condition instance could be returned as the value of a procedure, indicating that something unusual has happened, to allow the caller to clean up some state. The caller could then signal the condition once it is ready.

A more important reason for having a separate condition-signalling mechanism is that it allows *resignalling*. When a signalled condition has been caught by a particular handler, and the handler decides that it doesn't want to process that particular condition, it can signal the condition again. This is one way to allow other handlers to get a chance to see the condition.

`error` *reason argument ...* [procedure]

This is the simplest and most common way to signal a condition that requires intervention before a computation can proceed (when intervention is not required, `warn` is more appropriate). `error` signals a condition (using `signal-condition`), and if no handler for that condition alters the flow of control (by invoking a restart, for example) it calls the procedure `standard-error-handler`, which normally prints an error message and stops the computation, entering an error REPL. Under normal circumstances `error` will not return a value (although an interactive debugger can be used to force this to occur).

Precisely what condition is signalled depends on the first argument to `error`. If *reason* is a condition, then that condition is signalled and the *argument*s are ignored. If *reason* is a condition type, then a new instance of this type is generated and signalled; the *argument*s are used to generate the values of the fields for this condition type (they are passed as the *field-plist* argument to `make-condition`). In the most common case, however, *reason* is neither a condition nor a condition type, but rather a string or symbol. In this case a condition of type `condition-type:simple-error` is created with the *message* field containing the *reason* and the *irritants* field containing the *argument*s.

`warn` *reason argument ...* [procedure]

When a condition is not severe enough to warrant intervention, it is appropriate to signal the condition with `warn` rather than `error`. As with `error`, `warn` first calls `signal-condition`; the condition that is signalled is chosen exactly as in `error` except that a condition of type `condition-type:simple-warning` is signalled if *reason* is neither a condition nor a condition type. If the condition is not handled, `warn` calls the procedure `standard-warning-handler`, which normally prints a warning message and continues the computation by returning from `warn`.

`warn` establishes a restart named `muffle-warning` before calling `signal-condition`. This allows a signal handler to prevent the generation of the warning message by calling `muffle-warning`. The value of a call to `warn` is unspecified.

`signal-condition` *condition* [procedure]

This is the fundamental operation for signalling a condition. The precise operation of `signal-condition` depends on the condition type of which *condition* is an instance, the condition types set by `break-on-signals`, and the handlers established by `bind-condition-handler` and `bind-default-condition-handler`.

If the *condition* is an instance of a type that is a specialization of any of the types specified by `break-on-signals`, then a breakpoint REPL is initiated. Otherwise (or when that REPL returns), the handlers established by `bind-condition-handler` are checked, most recent first. Each applicable handler is invoked, and the search for a handler continues if the handler returns normally. If all applicable handlers return,

then the applicable handlers established by `bind-default-condition-handler` are checked, again most recent first. Finally, if no handlers apply (or all return in a normal manner), `signal-condition` returns an unspecified value.

Note: unlike many other systems, the MIT/GNU Scheme runtime library does *not* establish handlers of any kind. (However, the Edwin text editor uses condition handlers extensively.) Thus, calls to `signal-condition` will return to the caller unless there are user supplied condition handlers, as the following example shows:

```
(signal-condition
 (make-condition
  condition-type:error
  (call-with-current-continuation (lambda (x) x))
  '()     ; no restarts
  '()))   ; no fields
⇒  unspecified
```

16.2 Error Messages

By convention, error messages (and in general, the reports generated by `write-condition-report`) should consist of one or more complete sentences. The usual rules for sentences should be followed: the first word of the sentence should be capitalized, and the sentence should be terminated by a period. The message should not contain extraneous whitespace such as line breaks or indentation.

The error system provides a simple formatting language that allows the programmer to have some control over the printing of error messages. This formatting language will probably be redesigned in a future release.

Error messages typically consist of a string describing the error, followed by some irritant objects. The string is printed using `display`, and the irritants are printed using `write`, typically with a space between each irritant. To allow simple formatting, we introduce a *noise* object, printed using `display`. The irritant list may contain ordinary objects interspersed with noise objects. Each noise object is printed using `display`, with no extra whitespace, while each normal object is printed using `write`, prefixed by a single space character.

Here is an example:

```
(define (error-within-procedure message irritant procedure)
  (error message
         irritant
         (error-irritant/noise "within procedure")
         procedure
         (error-irritant/noise ".")))
```

This would format as follows:

```
(error-within-procedure "Bad widget" 'widget-32 'invert-widget)  error

Bad widget widget-32 within procedure invert-widget.
```

Here are the operations supporting error messages:

`format-error-message` *message irritants port* [procedure]

Message is typically a string (although this is not required), *irritants* a list of irritant objects, and *port* an output port. Formats *message* and *irritants* to *port* in the standard way. Note that, during the formatting process, the depth and breadth to which lists are printed are each limited to small numbers, to guarantee that the output from each irritant is not arbitrarily large.

`error-irritant/noise` *value* [procedure]

Creates and returns a noise object whose value is *value*.

16.3 Condition Handling

The occurrence of a condition is signalled using `signal-condition`. `signal-condition` attempts to locate and invoke a *condition handler* that is prepared to deal with the type of condition that has occurred. A condition handler is a procedure of one parameter, the condition that is being signalled. A procedure is installed as a condition handler by calling `bind-condition-handler` (to establish a handler that is in effect only while a particular thunk is executing) or `bind-default-condition-handler` (to establish a handler that is in effect permanently). As implied by the name, handlers created by `bind-default-condition-handler` are invoked only after all other applicable handlers have been invoked.

A *handler* may process a signal in any way it deems appropriate, but the common patterns are:

Ignore the condition.
: By returning from the handler in the usual manner.

Handle the condition.
: By doing some processing and then invoking a restart (or, less preferably, a continuation) that was established at some point prior to the call to `signal-condition`.

Resignal a condition.
: By doing some processing and calling `signal-condition` with either the same condition or a newly created one. In order to support this, `signal-condition` runs *handler* in such a way that a subsequent call to `signal-condition` sees only the handlers that were established prior to this one.

As an aid to debugging condition handlers, Scheme maintains a set of condition types that will cause an interactive breakpoint to occur prior to normal condition signalling. That is, `signal-condition` creates a new REPL prior to its normal operation when its argument is a condition that is a specialization of any of these types. The procedure `break-on-signals` establishes this set of condition types.

`ignore-errors` *thunk* [procedure]

Executes *thunk* with a condition handler that intercepts the signalling of any specialization of `condition-type:error` (including those produced by calls to `error`) and immediately terminates the execution of *thunk* and returns from the call to `ignore-errors` with the signalled condition as its value. If *thunk* returns normally, its value is returned from `ignore-errors`.

Notice that `ignore-errors` does not "turn off signalling" or condition handling. Condition handling takes place in the normal manner but conditions specialized from `condition-type:error` are trapped rather than propogated as they would be by default.

`bind-condition-handler` *condition-types handler thunk* [procedure]
Invokes *thunk* after adding *handler* as a condition handler for the conditions specified by *condition-types*. *Condition-types* must be a list of condition types; signalling a condition whose type is a specialization of any of these types will cause the *handler* to be invoked. See `signal-condition` for a description of the mechanism used to invoke handlers.

By special extension, if *condition-types* is the empty list then the *handler* is called for all conditions.

`bind-default-condition-handler` *condition-types handler* [procedure]
Installs *handler* as a (permanent) condition handler for the conditions specified by *condition-types*. *Condition-types* must be a list of condition types; signalling a condition whose type is a specialization of any of these types will cause the *handler* to be invoked. See `signal-condition` for a description of the mechanism used to invoke handlers.

By special extension, if *condition-types* is the empty list then the *handler* is called for all conditions.

`break-on-signals` *condition-types* [procedure]
Arranges for `signal-condition` to create an interactive REPL before it signals a condition that is a specialization of any of the types in the list of *condition-types*. This can be extremely helpful when trying to debug code that uses custom condition handlers. In order to create a REPL when *any* condition type is signalled it is best to actually put a breakpoint on entry to `signal-condition`.

`standard-error-handler` *condition* [procedure]
Called internally by `error` after it calls `signal-condition`. Normally creates a new REPL with the prompt `"error>"` (but see `standard-error-hook`). In order to simulate the effect of calling `error`, code may call `signal-condition` directly and then call `standard-error-handler` if `signal-condition` returns.

`standard-error-hook` [parameter]
This parameter controls the behavior of the procedure `standard-error-handler`, and hence `error`. It is intended to be bound with `parameterize` and is normally `#f`. It may be changed to a procedure of one argument and will then be invoked (with `standard-error-hook` rebound to `#f`) by `standard-error-handler` just prior to starting the error REPL. It is passed one argument, the condition being signalled.

`standard-warning-handler` *condition* [procedure]
This is the procedure called internally by `warn` after it calls `signal-condition`. The normal behavior of `standard-warning-handler` is to print a message (but see `standard-warning-hook`). More precisely, the message is printed to the port returned by `notification-output-port`. The message is formed by first printing the string

`"Warning: "` to this port, and then calling `write-condition-report` on *condition* and the port.

In order to simulate the effect of calling `warn`, code may call `signal-condition` directly and then call `standard-warning-handler` if `signal-condition` returns. (This is not sufficient to implement the `muffle-warning` protocol, however. For that purpose an explicit restart must be provided.)

`standard-warning-hook` [parameter]

This parameter controls the behavior of the procedure `standard-warning-handler`, and hence `warn`. It is intended to be bound with `parameterize` and is normally `#f`. It may be changed to a procedure of one argument and will then be invoked (with `standard-warning-hook` rebound to `#f`) by `standard-warning-handler` in lieu of writing the warning message. It is passed one argument, the condition being signalled.

16.4 Restarts

The Scheme error system provides a mechanism, known as *restarts*, that helps coordinate condition-signalling code with condition-handling code. A module of code that detects and signals conditions can provide procedures (using `with-simple-restart` or `with-restart`) to be invoked by handlers that wish to continue, abort, or restart the computation. These procedures, called *restart effectors*, are encapsulated in restart objects.

When a condition object is created, it contains a set of restart objects, each of which contains a restart effector. Condition handlers can inspect the condition they are handling (using `find-restart` to find restarts by name, or `condition/restarts` to see the entire set), and they can invoke the associated effectors (using `invoke-restart` or `invoke-restart-interactively`). Effectors can take arguments, and these may be computed directly by the condition-handling code or by gathering them interactively from the user.

The names of restarts can be chosen arbitrarily, but the choice of name is significant. These names are used to coordinate between the signalling code (which supplies names for restarts) and the handling code (which typically chooses a restart effector by the name of its restart). Thus, the names specify the *restart protocol* implemented by the signalling code and invoked by the handling code. The protocol indicates the number of arguments required by the effector code as well as the semantics of the arguments.

Scheme provides a conventional set of names (hence, protocols) for common use. By choosing the names of restarts from this set, signalling code can indicate that it is able to perform a small set of fairly common actions (`abort`, `continue`, `muffle-warning`, `retry`, `store-value`, `use-value`). In turn, simple condition-handling code can look for the kind of action it wishes to perform and simply invoke it by name. All of Scheme's conventional names are symbols, although in general restart names are not restricted to any particular data type. In addition, the object `#f` is reserved to indicate the "not for automated use" protocol: these restarts should be activated only under human control.

Restarts themselves are first-class objects. They encapsulate their name, a procedure (known as the *effector*) to be executed if they are invoked, and a thunk (known as the *reporter*) that can be invoked to display a description of the restart (used, for example, by the interactive debugger). Invoking a restart is an indication that a handler has chosen to accept control for a condition; as a consequence, the *effector* of the restart should not

return, since this would indicate that the handler declined to handle the condition. Thus, the *effector* should call a continuation captured before the condition-signalling process began. The most common pattern of usage by signalling code is encapsulated in `with-simple-restart`.

Within this chapter, a parameter named *restarts* will accept any of the following values:

- A list of restart objects.
- A condition. The procedure `condition/restarts` is called on the condition, and the resulting list of restarts is used in place of the condition.
- The symbol `bound-restarts`. The procedure `bound-restarts` is called (with no arguments), and the resulting list of restarts is used in place of the symbol.
- If the *restarts* parameter is optional and is not supplied, it is equivalent to having specified the symbol `bound-restarts`.

16.4.1 Establishing Restart Code

`with-simple-restart` *name reporter thunk* [procedure]

Invokes *thunk* in a dynamic environment created by adding a restart named *name* to the existing named restarts. *Reporter* may be used during the execution of *thunk* to produce a description of the newly created restart; it must either be a procedure of one argument (a port) or a string. By convention, the description generated by *reporter* should be a short complete sentence, with first word capitalized and terminated by a period. The sentence should fit on one line with a little room to spare (see the examples below); usually this means that the sentence should be 70 characters or less in length.

If the restart created by `with-simple-restart` is invoked it simply aborts the computation in progress by returning an unspecified value from the call to `with-simple-restart`. Otherwise `with-simple-restart` returns the value computed by *thunk*.

```
(with-simple-restart 'george "This restart is named george."
  (lambda () 3)) ⇒ 3

(with-simple-restart 'george "This restart is named george."
  (lambda ()
    (invoke-restart (find-restart 'george)))) ⇒ unspecific

(with-simple-restart 'george "This restart is named george."
  (lambda () (car 3)))
;The object 3, passed as the first argument to car,
; is not the correct type.
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify an argument to use in its place.
; (RESTART 2) => This restart is named george.
; (RESTART 1) => Return to read-eval-print level 1.
```

`with-restart` *name reporter effector interactor thunk* [procedure]

Invokes *thunk* in a dynamic environment created by adding a restart named *name* to the existing named restarts. *Reporter* may be used during the execution of *thunk*

to produce a description of the newly created restart; it must either be a procedure of one argument (a port) or a string. *Effector* is a procedure which will be called when the restart is invoked by `invoke-restart`. *Interactor* specifies the arguments that are to be passed to *effector* when it is invoked interactively; it may be either a procedure of no arguments, or `#f`. If *interactor* is `#f`, this restart is not meant to be invoked interactively.

The value returned by `with-restart` is the value returned by *thunk*. Should the restart be invoked by a condition handler, however, the *effector* will not return back to the handler that invoked it. Instead, the *effector* should call a continuation created before the condition-signalling process began, and `with-restart` will therefore not return in the normal manner.

```
(define (by-george! thunk)
  ; This code handles conditions that arise while executing thunk
  ; by invoking the GEORGE restart, passing 1 and 2 to the restart's
  ; effector code.
  (bind-condition-handler '() ; All conditions
   (lambda (condition)
     (invoke-restart (find-restart 'george) 1 2))
   thunk))

(define (can-george! thunk)
  ; This code provides a way of handling errors: the GEORGE restart.
  ; In order to GEORGE you must supply two values.
  (lambda ()
    (call-with-current-continuation
     (lambda (kappa)
       (with-restart
        'george                          ; Name
        "This restart is named george."  ; Reporter
        (lambda (a b)                    ; Effector
          (kappa (list 'george a b)))
        values                           ; Interactor
        thunk)))))                       ; Thunk

(by-george! (can-george! (lambda () -3))        ⇒ -3
(by-george! (can-george! (lambda () (car 'x)))) ⇒ (george 1 2)
```

16.4.2 Invoking Standard Restart Code

Scheme supports six standard protocols for restarting from a condition, each encapsulated using a named restart (for use by condition-signalling code) and a simple procedure (for use by condition-handling code). Unless otherwise specified, if one of these procedures is unable to find its corresponding restart, it returns immediately with an unspecified value.

Each of these procedures accepts an optional argument *restarts*, which is described above in Section 16.4 [Restarts], page 315.

`abort` [*restarts*] [procedure]
Abort the computation, using the restart named `abort`. The corresponding effector takes no arguments and abandons the current line of computation. This is the restart provided by Scheme's REPL.

If there is no restart named `abort`, this procedure signals an error of type `condition-type:no-such-restart`.

`continue` [*restarts*] [procedure]
Continue the current computation, using the restart named `continue`. The corresponding effector takes no arguments and continues the computation beyond the point at which the condition was signalled.

`muffle-warning` [*restarts*] [procedure]
Continue the current computation, using the restart named `muffle-warning`. The corresponding effector takes no arguments and continues the computation beyond the point at which any warning message resulting from the condition would be presented to the user. The procedure `warn` establishes a `muffle-warning` restart for this purpose.

If there is no restart named `muffle-warning`, this procedure signals an error of type `condition-type:no-such-restart`.

`retry` [*restarts*] [procedure]
Retry the current computation, using the restart named `retry`. The corresponding effector takes no arguments and simply retries the same computation that triggered the condition. The condition may reoccur, of course, if the root cause has not been eliminated. The code that signals a "file does not exist" error can be expected to supply a `retry` restart. The restart would be invoked after first creating the missing file, since the computation is then likely to succeed if it is simply retried.

`store-value` *new-value* [*restarts*] [procedure]
Retry the current computation, using the restart named `store-value`, after first storing *new-value*. The corresponding effector takes one argument, *new-value*, and stores it away in a restart-dependent location, then retries the same computation that triggered the condition. The condition may reoccur, of course, if the root cause has not been eliminated. The code that signals an "unassigned variable" error can be expected to supply a `store-value` restart; this would store the value in the variable and continue the computation.

`use-value` *new-value* [*restarts*] [procedure]
Retry the current computation, using the restart named `use-value`, but substituting *new-value* for a value that previously caused a failure. The corresponding effector takes one argument, *new-value*, and retries the same computation that triggered the condition with the new value substituted for the failing value. The condition may reoccur, of course, if the new value also induces the condition.

The code that signals an "unassigned variable" error can be expected to supply a `use-value` restart; this would simply continue the computation with *new-value* instead of the value of the variable. Contrast this with the `retry` and `store-value` restarts. If the `retry` restart is used it will fail because the variable still has no value. The

`store-value` restart could be used, but it would alter the value of the variable, so that future references to the variable would not be detected.

16.4.3 Finding and Invoking General Restart Code

Restarts are a general mechanism for establishing a protocol between condition-signalling and condition-handling code. The Scheme error system provides "packaging" for a number of common protocols. It also provides lower-level hooks that are intended for implementing customized protocols. The mechanism used by signalling code (`with-restart` and `with-simple-restart`) is used for both purposes.

Four additional operations are provided for the use of condition-handling code. Two operations (`bound-restarts` and `find-restart`) allow condition-handling code to locate active restarts. The other two operations (`invoke-restart` and `invoke-restart-interactively`) allow restart effectors to be invoked once the restart object has been located.

In addition, there is a data abstraction that provides access to the information encapsulated in restart objects.

`bound-restarts` [procedure]

Returns a list of all currently active restart objects, most recently installed first. `bound-restarts` should be used with caution by condition-handling code, since it reveals all restarts that are active at the time it is called, rather than at the time the condition was signalled. It is useful, however, for collecting the list of restarts for inclusion in newly generated condition objects or for inspecting the current state of the system.

`find-restart` *name* [*restarts*] [procedure]

Returns the first restart object named *name* in the list of *restarts* (permissible values for *restarts* are described above in Section 16.4 [Restarts], page 315). When used in a condition handler, `find-restart` is usually passed the name of a particular restart *and* the condition object that has been signalled. In this way the handler finds only restarts that were available when the condition was created (usually the same as when it was signalled). If *restarts* is omitted, the currently active restarts would be used, and these often include restarts added after the condition ocurred.

`invoke-restart` *restart argument* ... [procedure]

Calls the restart effector encapsulated in *restart*, passing the specified *argument*s to it. `invoke-restart` is intended for use by condition-handling code that understands the protocol implemented by *restart*, and can therefore calculate and pass an appropriate set of arguments.

If a condition handler needs to interact with a user to gather the arguments for an effector (e.g. if it does not understand the protocol implemented by *restart*) `invoke-restart-interactively` should be used instead of `invoke-restart`.

`invoke-restart-interactively` *restart* [procedure]

First calls the interactor encapsulated in *restart* to interactively gather the arguments needed for *restart*'s effector. It then calls the effector, passing these arguments to it.

`invoke-restart-interactively` is intended for calling interactive restarts (those for which `restart/interactor` is not `#f`). For convenience, `invoke-restart-interactively` will call the restart's effector with no arguments if the restart has no interactor; this behavior may change in the future.

16.4.4 The Named Restart Abstraction

A restart object is very simple, since it encapsulates only a name, effector, interactor, and description.

`restart?` *object* [procedure]
Returns `#f` if and only if *object* is not a restart.

`restart/name` *restart* [procedure]
Returns the name of *restart*. While the Scheme error system uses only symbols and the object `#f` for its predefined names, programs may use arbitrary objects (name equivalence is tested using `eq?`).

`restart/effector` *restart* [procedure]
Returns the effector encapsulated in *restart*. Normally this procedure is not used since `invoke-restart` and `invoke-restart-interactively` capture the most common invocation patterns.

`restart/interactor` *restart* [procedure]
Returns the interactor encapsulated in *restart*. This is either a procedure of no arguments or the object `#f`. Normally this procedure is not used since `invoke-restart-interactively` captures the most common usage. Thus `restart/interactor` is most useful as a predicate to determine if *restart* is intended to be invoked interactively.

`write-restart-report` *restart port* [procedure]
Writes a description of *restart* to *port*. This works by either displaying (if it is a string) or calling (if it is a procedure) the *reporter* that was supplied when the restart was created.

16.5 Condition Instances

A *condition*, in addition to the information associated with its type, usually contains other information that is not shared with other conditions of the same type. For example, the condition type associated with "unbound variable" errors does not specify the name of the variable that was unbound. The additional information is captured in a *condition* object, also called a *condition instance*.

In addition to information that is specific to a given type of condition (such as the variable name for "unbound variable" conditions), every condition instance also contains a continuation that encapsulates the state of the computation in which the condition occurred. This continuation is used for analyzing the computation to learn more about the context in which the condition occurred. It is *not* intended to provide a mechanism for continuing the computation; that mechanism is provided by restarts.

16.5.1 Generating Operations on Conditions

Scheme provides four procedures that take a condition type as input and produce operations on the corresponding condition object. These are reminiscent of the operations on record types that produce record operators (see Section 10.4 [Records], page 172). Given a condition type it is possible to generate: a constructor for instances of the type (using `condition-constructor`); an accessor to extract the contents of a field in instances of the type (using `condition-accessor`); a predicate to test for instances of the type (using `condition-predicate`); and a procedure to create and signal an instance of the type (using `condition-signaller`).

Notice that the creation of a condition object is distinct from signalling an occurrence of the condition. Condition objects are first-class; they may be created and never signalled, or they may be signalled more than once. Further notice that there are no procedures for modifying conditions; once created, a condition cannot be altered.

`condition-constructor` *condition-type field-names* [procedure]

Returns a constructor procedure that takes as arguments values for the fields specified in *field-names* and creates a condition of type *condition-type*. *Field-names* must be a list of symbols that is a subset of the *field-names* in *condition-type*. The constructor procedure returned by `condition-constructor` has signature

```
(lambda (continuation restarts . field-values) ...)
```

where the *field-names* correspond to the *field-values*. The constructor argument *restarts* is described in Section 16.4 [Restarts], page 315. Conditions created by the constructor procedure have `#f` for the values of all fields other than those specified by *field-names*.

For example, the following procedure `make-simple-warning` constructs a condition of type `condition-type:simple-warning` given a continuation (where the condition occurred), a description of the restarts to be made available, a warning message, and a list of irritants that caused the warning:

```
(define make-simple-warning
  (condition-constructor condition-type:simple-warning
                         '(message irritants)))
```

`condition-accessor` *condition-type field-name* [procedure]

Returns a procedure that takes as input a condition object of type *condition-type* and extracts the contents of the specified *field-name*. `condition-accessor` signals `error:bad-range-argument` if the *field-name* isn't one of the named fields of *condition-type*; the returned procedure will signal `error:wrong-type-argument` if passed an object other than a condition of type *condition-type* or one of its specializations.

If it is known in advance that a particular field of a condition will be accessed repeatedly it is worth constructing an accessor for the field using `condition-accessor` rather than using the (possibly more convenient, but slower) `access-condition` procedure.

`condition-predicate` *condition-type* [procedure]
Returns a predicate procedure for testing whether an object is a condition of type *condition-type* or one of its specializations (there is no predefined way to test for a condition of a given type but *not* a specialization of that type).

`condition-signaller` *condition-type field-names default-handler* [procedure]
Returns a signalling procedure with parameters *field-names*. When the signalling procedure is called it creates and signals a condition of type *condition-type*. If the condition isn't handled (i.e. if no handler is invoked that causes an escape from the current continuation) the signalling procedure reduces to a call to *default-handler* with the condition as its argument.

There are several standard procedures that are conventionally used for *default-handler*. If *condition-type* is a specialization of `condition-type:error`, *default-handler* should be the procedure `standard-error-handler`. If *condition-type* is a specialization of `condition-type:warning`, *default-handler* should be the procedure `standard-warning-handler`. If *condition-type* is a specialization of `condition-type:breakpoint`, *default-handler* should be the procedure `standard-breakpoint-handler`.

16.5.2 Condition Abstraction

The condition data type is abstracted through a predicate `condition?` and a set of accessor procedures.

`condition?` *object* [procedure]
Returns `#f` if and only if *object* is not a condition.

`condition/type` *condition* [procedure]
Returns the condition type of which *condition* is an instance.

`condition/error?` *condition* [procedure]
Returns `#t` if the *condition* is an instance of condition type `condition-type:error` or a specialization of it, `#f` otherwise.

`condition/restarts` *condition* [procedure]
Returns the list of restarts specified when *condition* was created.

`condition/continuation` *condition* [procedure]
Returns the continuation specified when *condition* was created. This is provided for inspecting the state of the system when the condition occurred, *not* for continuing or restarting the computation.

`write-condition-report` *condition port* [procedure]
Writes a description of *condition* to *port*, using the reporter function from the condition type associated with *condition*. See also `condition/report-string`.

16.5.3 Simple Operations on Condition Instances

The simple procedures described in this section are built on top of the more detailed abstraction of condition objects described above. While these procedures are sometimes easier to use, they are often less efficient.

`make-condition` *condition-type continuation restarts field-plist* [procedure]
Create a new condition object as an instance of *condition-type*, associated with *continuation*. The *continuation* is provided for inspection purposes only, *not* for restarting the computation. The *restarts* argument is described in Section 16.4 [Restarts], page 315. The *field-plist* is an alternating list of field names and values for those fields, where the field names are those that would be returned by `(condition-type/field-names` *`condition-type`*`)`. It is used to provide values for fields in the condition object; fields with no value specified are set to `#f`. Once a condition object has been created there is no way to alter the values of these fields.

`access-condition` *condition field-name* [procedure]
Returns the value stored in the field *field-name* within *condition*. *Field-name* must be one of the names returned by `(condition-type/field-names (condition/type` *`condition`*`))`. `access-condition` looks up the *field-name* at runtime, so it is more efficient to use `condition-accessor` to create an access function if the same field is to be extracted from several instances of the same condition type.

`condition/report-string` *condition* [procedure]
Returns a string containing a report of the *condition*. This is generated by calling `write-condition-report` on *condition* and a string output port, and returning the output collected by the port as a string.

16.6 Condition Types

Each condition has a *condition type* object associated with it. These objects are used as a means of focusing on related classes of conditions, first by concentrating all of the information about a specific class of condition in a single place, and second by specifying an inheritance relationship between types. This inheritance relationship forms the taxonomic structure of the condition hierarchy (see Section 16.7 [Taxonomy], page 324).

The following procedures consititute the abstraction for condition types.

`make-condition-type` *name generalization field-names reporter* [procedure]
Creates and returns a (new) condition type that is a specialization of *generalization* (if it is a condition type) or is the root of a new tree of condition types (if *generalization* is `#f`). For debugging purposes, the condition type has a *name*, and instances of this type contain storage for the fields specified by *field-names* (a list of symbols) in addition to the fields common to all conditions (*type*, *continuation* and *restarts*).

Reporter is used to produce a description of a particular condition of this type. It may be a string describing the condition, a procedure of arity two (the first argument will be a condition of this type and the second a port) that will `write` the message to the given port, or `#f` to specify that the reporter should be taken from the condition type *generalization* (or produce an "undocumented condition of type ..." message if *generalization* is `#f`). The conventions used to form descriptions are spelled out in Section 16.2 [Error Messages], page 312.

`condition-type/error?` *condition-type* [procedure]
Returns `#t` if the *condition-type* is `condition-type:error` or a specialization of it, `#f` otherwise.

`condition-type/field-names` *condition-type* [procedure]
Returns a list of all of the field names for a condition of type *condition-type*. This is the set union of the fields specified when this *condition-type* was created with the `condition-type/field-names` of the generalization of this *condition-type*.

`condition-type/generalizations` *condition-type* [procedure]
Returns a list of all of the generalizations of *condition-type*. Notice that every condition type is considered a generalization of itself.

`condition-type?` *object* [procedure]
Returns `#f` if and only if *object* is not a condition type.

16.7 Condition-Type Taxonomy

The MIT/GNU Scheme error system provides a rich set of predefined condition types. These are organized into a forest through taxonomic links providing the relationships for "specializes" and "generalizes". The chart appearing below shows these relationships by indenting all the specializations of a given type relative to the type. Note that the variables that are bound to these condition types are prefixed by '`condition-type:`'; for example, the type appearing in the following table as '`simple-error`' is stored in the variable `condition-type:simple-error`. Users are encouraged to add new condition types by creating specializations of existing ones.

Following the chart are detailed descriptions of the predefined condition types. Some of these types are marked as *abstract* types. Abstract types are not intended to be used directly as the type of a condition; they are to be used as generalizations of other types, and for binding condition handlers. Types that are not marked as abstract are *concrete*; they are intended to be explicitly used as a condition's type.

```
serious-condition
    error
        simple-error
        illegal-datum
            wrong-type-datum
                wrong-type-argument
                wrong-number-of-arguments
            datum-out-of-range
                bad-range-argument
            inapplicable-object
        file-error
            file-operation-error
            derived-file-error
        port-error
            derived-port-error
        variable-error
            unbound-variable
            unassigned-variable
        arithmetic-error
            divide-by-zero
            floating-point-overflow
            floating-point-underflow
        control-error
            no-such-restart
        not-loading
        primitive-procedure-error
            system-call-error
warning
    simple-warning
simple-condition
breakpoint
```

`condition-type:serious-condition` [condition type]
This is an abstract type. All serious conditions that require some form of intervention should inherit from this type. In particular, all errors inherit from this type.

`condition-type:error` [condition type]
This is an abstract type. All errors should inherit from this type.

`condition-type:simple-error` *message irritants* [condition type]
This is the condition generated by the `error` procedure when its first argument is not a condition or condition type. The fields *message* and *irritants* are taken directly from the arguments to `error`; *message* contains an object (usually a string) and *irritants* contains a list of objects. The reporter for this type uses `format-error-message` to generate its output from *message* and *irritants*.

`condition-type:illegal-datum` *datum* [condition type]
This is an abstract type. This type indicates the class of errors in which a program discovers an object that lacks specific required properties. Most commonly, the object

is of the wrong type or is outside a specific range. The *datum* field contains the offending object.

`condition-type:wrong-type-datum` *datum type* [condition type]
This type indicates the class of errors in which a program discovers an object that is of the wrong type. The *type* field contains a string describing the type that was expected, and the *datum* field contains the object that is of the wrong type.

```
(error:wrong-type-datum 3.4 "integer")   error
;The object 3.4 is not an integer.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:wrong-type-datum` *datum type* [procedure]
This procedure signals a condition of type `condition-type:wrong-type-datum`. The *datum* and *type* fields of the condition are filled in from the corresponding arguments to the procedure.

`condition-type:wrong-type-argument` *datum type operator operand* [condition type]
This type indicates that a procedure was passed an argument of the wrong type. The *operator* field contains the procedure (or a symbol naming the procedure), the *operand* field indicates the argument position that was involved (this field contains either a symbol, a non-negative integer, or `#f`), the *type* field contains a string describing the type that was expected, and the *datum* field contains the offending argument.

```
(+ 'a 3)                                   error
;The object a, passed as the first argument to integer-add,
; is not the correct type.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify an argument to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.

(list-copy 3)
;The object 3, passed as an argument to list-copy, is not a list.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:wrong-type-argument` *datum type operator* [procedure]
This procedure signals a condition of type `condition-type:wrong-type-argument`. The *datum*, *type* and *operator* fields of the condition are filled in from the corresponding arguments to the procedure; the *operand* field of the condition is set to `#f`.

`condition-type:wrong-number-of-arguments` *datum type operands* [condition type]
This type indicates that a procedure was called with the wrong number of arguments. The *datum* field contains the procedure being called, the *type* field contains the number of arguments that the procedure accepts, and the *operands* field contains a list of the arguments that were passed to the procedure.

```
(car 3 4)                                        error
;The procedure car has been called with 2 arguments;
; it requires exactly 1 argument.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:wrong-number-of-arguments` *datum type operands* [procedure]
This procedure signals a condition of type `condition-type:wrong-number-of-arguments`. The *datum*, *type* and *operands* fields of the condition are filled in from the corresponding arguments to the procedure.

`condition-type:datum-out-of-range` *datum* [condition type]
This type indicates the class of errors in which a program discovers an object that is of the correct type but is otherwise out of range. Most often, this type indicates that an index to some data structure is outside of the range of indices for that structure. The *datum* field contains the offending object.

```
(error:datum-out-of-range 3)                     error
;The object 3 is not in the correct range.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:datum-out-of-range` *datum* [procedure]
This procedure signals a condition of type `condition-type:datum-out-of-range`. The *datum* field of the condition is filled in from the corresponding argument to the procedure.

`condition-type:bad-range-argument` *datum operator operand* [condition type]
This type indicates that a procedure was passed an argument that is of the correct type but is otherwise out of range. Most often, this type indicates that an index to some data structure is outside of the range of indices for that structure. The *operator* field contains the procedure (or a symbol naming the procedure), the *operand* field indicates the argument position that was involved (this field contains either a symbol, a non-negative integer, or `#f`), and the *datum* field is the offending argument.

```
(string-ref "abc" 3)                             error
;The object 3, passed as the second argument to string-ref,
; is not in the correct range.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify an argument to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:bad-range-argument` *datum operator* [procedure]
This procedure signals a condition of type `condition-type:bad-range-argument`. The *datum* and *operator* fields of the condition are filled in from the corresponding arguments to the procedure; the *operand* field of the condition is set to `#f`.

`condition-type:inapplicable-object` *datum operands* [condition type]
This type indicates an error in which a program attempted to apply an object that is not a procedure. The object being applied is saved in the *datum* field, and the arguments being passed to the object are saved as a list in the *operands* field.

```
(3 4)                                              error
;The object 3 is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

`condition-type:file-error` *filename* [condition type]
This is an abstract type. It indicates that an error associated with a file has occurred. For example, attempting to delete a nonexistent file will signal an error. The *filename* field contains a filename or pathname associated with the operation that failed.

`condition-type:file-operation-error` *filename verb noun reason operator operands* [condition type]
This is the most common condition type for file system errors. The *filename* field contains the filename or pathname that was being operated on. The *verb* field contains a string which is the verb or verb phrase describing the operation being performed, and the *noun* field contains a string which is a noun or noun phrase describing the object being operated on. The *reason* field contains a string describing the error that occurred. The *operator* field contains the procedure performing the operation (or a symbol naming that procedure), and the *operands* field contains a list of the arguments that were passed to that procedure. For example, an attempt to delete a nonexistent file would have the following field values:

```
filename        "/zu/cph/tmp/no-such-file"
verb            "delete"
noun            "file"
reason          "no such file or directory"
operator        file-remove
operands        ("/zu/cph/tmp/no-such-file")
```

and would generate a message like this:

```
(delete-file "/zu/cph/tmp/no-such-file")  error
;Unable to delete file "/zu/cph/tmp/no-such-file" because:
; No such file or directory.
;To continue, call RESTART with an option number:
; (RESTART 3) => Try to delete the same file again.
; (RESTART 2) => Try to delete a different file.
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:file-operation` *index verb noun reason operator operands* [procedure]
This procedure signals a condition of type `condition-type:file-operation-error`. The fields of the condition are filled in from the corresponding arguments to the procedure, except that the filename is taken as the *index*th element of *operands*.

`condition-type:derived-file-error` *filename condition* [condition type]
This is another kind of file error, which is generated by obscure file-system errors that do not fit into the standard categories. The *filename* field contains the filename or pathname that was being operated on, and the *condition* field contains a condition describing the error in more detail. Usually the *condition* field contains a condition of type `condition-type:system-call-error`.

`error:derived-file` *filename condition* [procedure]
This procedure signals a condition of type `condition-type:derived-file-error`. The *filename* and *condition* fields of the condition are filled in from the corresponding arguments to the procedure.

`condition-type:port-error` *port* [condition type]
This is an abstract type. It indicates that an error associated with a I/O port has occurred. For example, writing output to a file port can signal an error if the disk containing the file is full; that error would be signalled as a port error. The *port* field contains the associated port.

`condition-type:derived-port-error` *port condition* [condition type]
This is a concrete type that is signalled when port errors occur. The *port* field contains the port associated with the error, and the *condition* field contains a condition object that describes the error in more detail. Usually the *condition* field contains a condition of type `condition-type:system-call-error`.

`error:derived-port` *port condition* [procedure]
This procedure signals a condition of type `condition-type:derived-port-error`. The *port* and *condition* fields of the condition are filled in from the corresponding arguments to the procedure.

`condition-type:variable-error` *location environment* [condition type]
This is an abstract type. It indicates that an error associated with a variable has occurred. The *location* field contains the name of the variable, and the *environment* field contains the environment in which the variable was referenced.

`condition-type:unbound-variable` *location environment* [condition type]
This type is generated when a program attempts to access or modify a variable that is not bound. The *location* field contains the name of the variable, and the *environment* field contains the environment in which the reference occurred.

```
foo                                                    error
;Unbound variable: foo
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of foo.
; (RESTART 2) => Define foo to a given value.
; (RESTART 1) => Return to read-eval-print level 1.
```

`condition-type:unassigned-variable` *location environment* [condition type]
This type is generated when a program attempts to access a variable that is not assigned. The *location* field contains the name of the variable, and the *environment* field contains the environment in which the reference occurred.

```
foo                                              error
;Unassigned variable: foo
;To continue, call RESTART with an option number:
; (RESTART 3) => Specify a value to use instead of foo.
; (RESTART 2) => Set foo to a given value.
; (RESTART 1) => Return to read-eval-print level 1.
```

`condition-type:arithmetic-error` *operator operands* [condition type]
This is an abstract type. It indicates that a numerical operation was unable to complete because of an arithmetic error. (For example, division by zero.) The *operator* field contains the procedure that implements the operation (or a symbol naming the procedure), and the *operands* field contains a list of the arguments that were passed to the procedure.

`condition-type:divide-by-zero` *operator operands* [condition type]
This type is generated when a program attempts to divide by zero. The *operator* field contains the procedure that implements the failing operation (or a symbol naming the procedure), and the *operands* field contains a list of the arguments that were passed to the procedure.

```
(/ 1 0)
;Division by zero signalled by /.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:divide-by-zero` *operator operands* [procedure]
This procedure signals a condition of type `condition-type:divide-by-zero`. The *operator* and *operands* fields of the condition are filled in from the corresponding arguments to the procedure.

`condition-type:floating-point-overflow` *operator operands* [condition type]
This type is generated when a program performs an arithmetic operation that results in a floating-point overflow. The *operator* field contains the procedure that implements the operation (or a symbol naming the procedure), and the *operands* field contains a list of the arguments that were passed to the procedure.

`condition-type:floating-point-underflow` *operator operands* [condition type]
This type is generated when a program performs an arithmetic operation that results in a floating-point underflow. The *operator* field contains the procedure that implements the operation (or a symbol naming the procedure), and the *operands* field contains a list of the arguments that were passed to the procedure.

`condition-type:primitive-procedure-error` *operator operands* [condition type]
This is an abstract type. It indicates that an error was generated by a primitive procedure call. Primitive procedures are distinguished from ordinary procedures in that they are not written in Scheme but instead in the underlying language of the Scheme implementation. The *operator* field contains the procedure that implements

the operation (or a symbol naming the procedure), and the *operands* field contains a list of the arguments that were passed to the procedure.

`condition-type:system-call-error` *operator operands system-call error-type* [condition type]

This is the most common condition type generated by primitive procedures. A condition of this type indicates that the primitive made a system call to the operating system, and that the system call signalled an error. The system-call error is reflected back to Scheme as a condition of this type, except that many common system-call errors are automatically translated by the Scheme implementation into more useful forms; for example, a system-call error that occurs while trying to delete a file will be translated into a condition of type `condition-type:file-operation-error`. The *operator* field contains the procedure that implements the operation (or a symbol naming the procedure), and the *operands* field contains a list of the arguments that were passed to the procedure. The *system-call* and *error-type* fields contain symbols that describe the specific system call that was being made and the error that occurred, respectively; these symbols are completely operating-system dependent.

`condition-type:control-error` [condition type]

This is an abstract type. It describes a class of errors relating to program control flow.

`condition-type:no-such-restart` *name* [condition type]

This type indicates that a named restart was not active when it was expected to be. Conditions of this type are signalled by several procedures that look for particular named restarts, for example `muffle-warning`. The *name* field contains the name that was being searched for.

```
(muffle-warning)                                    error
;The restart named muffle-warning is not bound.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`error:no-such-restart` *name* [procedure]

This procedure signals a condition of type `condition-type:no-such-restart`. The *name* field of the condition is filled in from the corresponding argument to the procedure.

`condition-type:not-loading` [condition type]

A condition of this type is generated when the procedure `current-load-pathname` is called from somewhere other than inside a file being loaded.

```
(current-load-pathname)                             error
;No file being loaded.
;To continue, call RESTART with an option number:
; (RESTART 1) => Return to read-eval-print level 1.
```

`condition-type:warning` [condition type]

This is an abstract type. All warnings should inherit from this type. Warnings are a class of conditions that are usually handled by informing the user of the condition and proceeding the computation normally.

`condition-type:simple-warning` *message irritants* [condition type]
This is the condition generated by the `warn` procedure. The fields *message* and *irritants* are taken directly from the arguments to `warn`; *message* contains an object (usually a string) and *irritants* contains a list of objects. The reporter for this type uses `format-error-message` to generate its output from *message* and *irritants*.

`condition-type:simple-condition` *message irritants* [condition type]
This is an unspecialized condition that does not fall into any of the standard condition classes. The *message* field contains an object (usually a string) and *irritants* contains a list of objects. The reporter for this type uses `format-error-message` to generate its output from *message* and *irritants*.

`condition-type:breakpoint` *environment message prompt* [condition type]
A condition of this type is generated by the breakpoint mechanism. The contents of its fields are beyond the scope of this document.

17 Graphics

MIT/GNU Scheme has a simple two-dimensional line-graphics interface that is suitable for many graphics applications. In particular it is often used for plotting data points from experiments. The interface is generic in that it can support different types of graphics devices in a uniform manner. At the present time only one type of graphics device is implemented on each operating system.

Procedures are available for drawing points, lines, and text; defining the coordinate system; clipping graphics output; controlling some of the drawing characteristics; and controlling the output buffer (for devices that perform buffering). Additionally, devices may support custom operations, such as control of colors.

There are some constraints on the arguments to the procedures described in this chapter. Any argument named *graphics-device* must be a graphics device object that was returned from a call to `make-graphics-device`. Any argument that is a coordinate must be either an exact integer or an inexact real.

17.1 Opening and Closing of Graphics Devices

`graphics-type-available?` *graphics-device-type* [procedure]
: This predicate returns `#t` if the graphics system named by the symbol *graphics-device-type* is implemented by the Scheme system. Otherwise it returns `#f`, in which case it is an error to attempt to make a graphics device using *graphics-device-type*.

`enumerate-graphics-types` [procedure]
: This procedure returns a list of symbols which are the names of all the graphics device types that are supported by the Scheme system. The result is useful in deciding what additional arguments to supply to `make-graphics-device`, as each device type typically has a unique way of specifying the initial size, shape and other attributes.

`make-graphics-device` *graphics-device-type object ...* [procedure]
: This operation creates and returns a graphics device object. *Graphics-device-type* is a symbol naming a graphics device type, and both the number and the meaning of the remaining arguments is determined by that type (see the description of each device type for details); *graphics-device-type* must satisfy `graphics-type-available?`. *Graphics-device-type* may also be `#f`, in which case the graphics device type is chosen by the system from what is available. This allows completely portable graphics programs to be written provided no custom graphics operations are used. When *graphics-device-type* is `#f` no further arguments may be given; each graphics device type will use some "sensible" defaults. If more control is required then the program should use one of the two procedures above to dispatch on the available types.

  This procedure opens and initializes the device, which remains valid until explicitly closed by the procedure `graphics-close`. Depending on the implementation of the graphics device, if this object is reclaimed by the garbage collector, the graphics device may remain open or it may be automatically closed. While a graphics device remains open the resources associated with it are not released.

`graphics-close` *graphics-device* [procedure]
Closes *graphics-device*, releasing its resources. Subsequently it is an error to use *graphics-device*.

17.2 Coordinates for Graphics

Each graphics device has two different coordinate systems associated with it: *device coordinates* and *virtual coordinates*. Device coordinates are generally defined by low-level characteristics of the device itself, and often cannot be changed. Most device coordinate systems are defined in terms of pixels, and usually the upper-left-hand corner is the origin of the coordinate system, with x coordinates increasing to the right and y coordinates increasing downwards.

In contrast, virtual coordinates are more flexible in the units employed, the position of the origin, and even the direction in which the coordinates increase. A virtual coordinate system is defined by assigning coordinates to the edges of a device. Because these edge coordinates are arbitrary real numbers, any Cartesian coordinate system can be defined.

All graphics procedures that use coordinates are defined on virtual coordinates. For example, to draw a line at a particular place on a device, the virtual coordinates for the endpoints of that line are given.

When a graphics device is initialized, its virtual coordinate system is reset so that the left edge corresponds to an x-coordinate of `-1`, the right edge to x-coordinate `1`, the bottom edge to y-coordinate `-1`, and the top edge to y-coordinate `1`.

`graphics-device-coordinate-limits` *graphics-device* [procedure]
Returns (as multiple values) the device coordinate limits for *graphics-device*. The values, which are exact non-negative integers, are: *x-left*, *y-bottom*, *x-right*, and *y-top*.

`graphics-coordinate-limits` *graphics-device* [procedure]
Returns (as multiple values) the virtual coordinate limits for *graphics-device*. The values, which are real numbers, are: *x-left*, *y-bottom*, *x-right*, and *y-top*.

`graphics-set-coordinate-limits` *graphics-device x-left y-bottom x-right y-top* [procedure]
Changes the virtual coordinate limits of *graphics-device* to the given arguments. *X-left*, *y-bottom*, *x-right*, and *y-top* must be real numbers. Subsequent calls to `graphics-coordinate-limits` will return the new limits. This operation has no effect on the device's displayed contents.

Note: This operation usually resets the clip rectangle, although it is not guaranteed to do so. If a clip rectangle is in effect when this procedure is called, it is necessary to redefine the clip rectangle afterwards.

17.3 Drawing Graphics

The procedures in this section provide the basic drawing capabilities of Scheme's graphics system.

`graphics-clear` *graphics-device* [procedure]
Clears the display of *graphics-device*. Unaffected by the current drawing mode.

`graphics-draw-point` *graphics-device x y* [procedure]
Draws a single point on *graphics-device* at the virtual coordinates given by *x* and *y*, using the current drawing mode.

`graphics-erase-point` *graphics-device x y* [procedure]
Erases a single point on *graphics-device* at the virtual coordinates given by *x* and *y*. This procedure is unaffected by the current drawing mode.

This is equivalent to

```
(lambda (device x y)
  (graphics-bind-drawing-mode device 0
    (lambda ()
      (graphics-draw-point device x y))))
```

`graphics-draw-line` *graphics-device x-start y-start x-end y-end* [procedure]
X-start, *y-start*, *x-end*, and *y-end* must be real numbers. Draws a line on *graphics-device* that connects the points (*x-start*, *y-start*) and (*x-end*, *y-end*). The line is drawn using the current drawing mode and line style.

`graphics-draw-text` *graphics-device x y string* [procedure]
Draws the characters of *string* at the point (*x*, *y*) on *graphics-device*, using the current drawing mode. The characteristics of the characters drawn are device-dependent, but all devices are initialized so that the characters are drawn upright, from left to right, with the leftmost edge of the leftmost character at *x*, and the baseline of the characters at *y*.

The following two procedures provide an alternate mechanism for drawing lines, which is more akin to using a plotter. They maintain a *cursor*, which can be positioned to a particular point and then dragged to another point, producing a line. Sequences of connected line segments can be drawn by dragging the cursor from point to point.

Many graphics operations have an unspecified effect on the cursor. The following exceptions are guaranteed to leave the cursor unaffected:

```
graphics-device-coordinate-limits
graphics-coordinate-limits
graphics-enable-buffering
graphics-disable-buffering
graphics-flush
graphics-bind-drawing-mode
graphics-set-drawing-mode
graphics-bind-line-style
graphics-set-line-style
```

The initial state of the cursor is unspecified.

`graphics-move-cursor` *graphics-device x y* [procedure]
Moves the cursor for *graphics-device* to the point (*x*, *y*). The contents of the device's display are unchanged.

`graphics-drag-cursor` *graphics-device x y* [procedure]
Draws a line from *graphics-device*'s cursor to the point (*x*, *y*), simultaneously moving the cursor to that point. The line is drawn using the current drawing mode and line style.

17.4 Characteristics of Graphics Output

Two characteristics of graphics output are so useful that they are supported uniformly by all graphics devices: *drawing mode* and *line style*. A third characteristic, *color*, is equally useful (if not more so), but implementation restrictions prohibit a uniform interface.

The *drawing mode*, an exact integer in the range `0` to `15` inclusive, determines how the figure being drawn is combined with the background over which it is drawn to generate the final result. Initially the drawing mode is set to "source", so that the new output overwrites whatever appears in that place. Useful alternative drawing modes can, for example, erase what was already there, or invert it.

Altogether 16 boolean operations are available for combining the source (what is being drawn) and the destination (what is being drawn over). The source and destination are combined by the device on a pixel-by-pixel basis as follows:

Mode	Meaning
0	ZERO [erase; use background color]
1	source AND destination
2	source AND (NOT destination)
3	source
4	(NOT source) AND destination
5	destination
6	source XOR destination
7	source OR destination
8	NOT (source OR destination)
9	NOT (source XOR destination)
10	NOT destination
11	source OR (NOT destination)
12	NOT source
13	(NOT source) OR destination
14	(NOT source) OR (NOT destination)
15	ONE [use foreground color]

The *line style*, an exact integer in the range `0` to `7` inclusive, determines which parts of a line are drawn in the foreground color, and which in the background color. The default line style, "solid", draws the entire line in the foreground color. Alternatively, the "dash" style alternates between foreground and background colors to generate a dashed line. This capability is useful for plotting several things on the same graph.

Here is a table showing the name and approximate pattern of the different styles. A '`1`' in the pattern represents a foreground pixel, while a '`-`' represents a background pixel. Note that the precise output for each style will vary from device to device. The only style that is guaranteed to be the same for every device is "solid".

```
Style   Name                    Pattern
-----   -------                 -------
0       solid                   1111111111111111
1       dash                    11111111--------
2       dot                     1-1-1-1-1-1-1-1-
3       dash dot                1111111111111-1-
4       dash dot dot            11111111111-1-1-
5       long dash               11111111111-----
6       center dash             111111111111-11-
7       center dash dash        111111111-11-11-
```

`graphics-bind-drawing-mode` *graphics-device drawing-mode thunk* [procedure]
`graphics-bind-line-style` *graphics-device line-style thunk* [procedure]
These procedures bind the drawing mode or line style, respectively, of *graphics-device*, invoke the procedure *thunk* with no arguments, then undo the binding when *thunk* returns. The value of each procedure is the value returned by *thunk*. Graphics operations performed during *thunk*'s dynamic extent will see the newly bound mode or style as current.

`graphics-set-drawing-mode` *graphics-device drawing-mode* [procedure]
`graphics-set-line-style` *graphics-device line-style* [procedure]
These procedures change the drawing mode or line style, respectively, of *graphics-device*. The mode or style will remain in effect until subsequent changes or bindings.

17.5 Buffering of Graphics Output

To improve performance of graphics output, most graphics devices provide some form of buffering. By default, Scheme's graphics procedures flush this buffer after every drawing operation. The procedures in this section allow the user to control the flushing of the output buffer.

`graphics-enable-buffering` *graphics-device* [procedure]
Enables buffering for *graphics-device*. In other words, after this procedure is called, graphics operations are permitted to buffer their drawing requests. This usually means that the drawing is delayed until the buffer is flushed explicitly by the user, or until it fills up and is flushed by the system.

`graphics-disable-buffering` *graphics-device* [procedure]
Disables buffering for *graphics-device*. By default, all graphics devices are initialized with buffering disabled. After this procedure is called, all drawing operations perform their output immediately, before returning.

Note: `graphics-disable-buffering` flushes the output buffer if necessary.

`graphics-flush` *graphics-device* [procedure]
Flushes the graphics output buffer for *graphics-device*. This operation has no effect for devices that do not support buffering, or if buffering is disabled for the device.

17.6 Clipping of Graphics Output

Scheme provides a rudimentary mechanism for restricting graphics output to a given rectangular subsection of a graphics device. By default, graphics output that is drawn anywhere within the device's virtual coordinate limits will appear on the device. When a *clip rectangle* is specified, however, output that would have appeared outside the clip rectangle is not drawn.

Note that changing the virtual coordinate limits for a device will usually reset the clip rectangle for that device, as will any operation that affects the size of the device (such as a window resizing operation). However, programs should not depend on this.

`graphics-set-clip-rectangle` *graphics-device x-left y-bottom x-right y-top* [procedure]

Specifies the clip rectangle for *graphics-device* in virtual coordinates. *X-left*, *y-bottom*, *x-right*, and *y-top* must be real numbers. Subsequent graphics output is clipped to the intersection of this rectangle and the device's virtual coordinate limits.

`graphics-reset-clip-rectangle` *graphics-device* [procedure]

Eliminates the clip rectangle for *graphics-device*. Subsequent graphics output is clipped to the virtual coordinate limits of the device.

17.7 Custom Graphics Operations

In addition to the standard operations, a graphics device may support *custom operations*. For example, most devices have custom operations to control color. `graphics-operation` is used to invoke custom operations.

`graphics-operation` *graphics-device name object ...* [procedure]

Invokes the graphics operation on *graphics-device* whose name is the symbol *name*, passing it the remaining arguments. This procedure can be used to invoke the standard operations, as well as custom operations that are specific to a particular graphics device type. The names of the standard graphics operations are formed by removing the `graphics-` prefix from the corresponding procedure. For example, the following are equivalent:

```
(graphics-draw-point device x y)
(graphics-operation device 'draw-point x y)
```

For information on the custom operations for a particular device, see the documentation for its type.

17.8 Images

Some graphics device types support images, which are rectangular pieces of picture that may be drawn into a graphics device. Images are often called something else in the host graphics system, such as bitmaps or pixmaps. The operations supported vary between devices, so look under the different device types to see what operations are available. All devices that support images support the following operations.

`create-image` *width height* [operation on `graphics-device`]

Images are created using the `create-image` graphics operation, specifying the *width* and *height* of the image in device coordinates (pixels).

```
(graphics-operation device 'create-image 200 100)
```

The initial contents of an image are unspecified.

`create-image` is a graphics operation rather than a procedure because the kind of image returned depends on the kind of graphics device used and the options specified in its creation. The image may be used freely with other graphics devices created with the same attributes, but the effects of using an image with a graphics device with different attributes (for example, different colors) is undefined. Under X, the image is display dependent.

`draw-image` *x y image* [operation on `graphics-device`]
The image is copied into the graphics device at the specified position.

`draw-subimage` *x y image im-x im-y w h* [operation on `graphics-device`]
Part of the image is copied into the graphics device at the specified (*x*, *y*) position. The part of the image that is copied is the rectangular region at *im-x* and *im-y* and of width *w* and height *h*. These four numbers are given in device coordinates (pixels).

`image?` *object* [procedure]
Returns `#t` if *object* is an image, otherwise returns `#f`.

`image/destroy` *image* [procedure]
This procedure destroys *image*, returning storage to the system. Programs should destroy images after they have been used because even modest images may use large amounts of memory. Images are reclaimed by the garbage collector, but they may be implemented using memory outside of Scheme's heap. If an image is reclaimed before being destroyed, the implementation might not deallocate that non-heap memory, which can cause a subsequent call to `create-image` to fail because it is unable to allocate enough memory.

`image/height` *image* [procedure]
Returns the height of the image in device coordinates.

`image/width` *image* [procedure]
Returns the width of the image in device coordinates.

`image/fill-from-byte-vector` *image bytes* [procedure]
The contents of *image* are set in a device-dependent way, using one byte per pixel from *bytes* (a string). Pixels are filled row by row from the top of the image to the bottom, with each row being filled from left to right. There must be at least `(* (image/height` *image*`) (image/width` *image*`))` bytes in *bytes*.

17.9 X Graphics

MIT/GNU Scheme supports graphics in the X window system (version 11). Arbitrary numbers of displays may be opened, and arbitrary numbers of graphics windows may be created for each display. A variety of operations is available to manipulate various aspects of the windows, to control their size, position, colors, and mapping. The X graphics device type supports images, which are implemented as Xlib `XImage` objects. X display, window, and image objects are automatically closed if they are reclaimed by the garbage collector.

17.9.1 X Graphics Type

A graphics device for X windows is created by passing the symbol `x` as the graphics device type name to `make-graphics-device`:

```
(make-graphics-device 'x #!optional display geometry suppress-map?)
```

where *display* is either a display object, `#f`, or a string; *geometry* is either `#f` or a string; and *suppress-map?* is a boolean or a vector (see below). A new window is created on the appropriate display, and a graphics device representing that window is returned.

Display specifies which X display the window is to be opened on; if it is `#f` or a string, it is passed as an argument to `x-open-display`, and the value returned by that procedure is used in place of the original argument. *Geometry* is an X geometry string, or `#f` which means to use the default geometry (which is specified as a resource).

Suppress-map?, if given, may take two forms. First, it may be a boolean: if `#f` (the default), the window is automatically mapped after it is created; otherwise, `#t` means to suppress this automatic mapping. The second form is a vector of three elements. The first element is a boolean with the same meaning as the boolean form of *suppress-map?*. The second element is a string, which specifies an alternative resource name to be used for looking up the window's resources. The third element is also a string, which specifies a class name for looking up the window's resources. The default value for *suppress-map?* is `#f`.

The default resource and class names are `"schemeGraphics"` and `"SchemeGraphics"` respectively.

The window is initialized using the resource and class names specified by *suppress-map?*, and is sensitive to the following resource properties:

Property	Class	Default
geometry	Geometry	512x384+0+0
font	Font	fixed
borderWidth	BorderWidth	2
internalBorder	BorderWidth	[border width]
background	Background	white
foreground	Foreground	black
borderColor	BorderColor	[foreground color]
cursorColor	Foreground	[foreground color]
pointerColor	Foreground	[foreground color]

The window is created with a `backing_store` attribute of `Always`. The window's name and icon name are initialized to `"scheme-graphics"`.

17.9.2 Utilities for X Graphics

`x-graphics/open-display` *display-name* [procedure]

Opens a connection to the display whose name is *display-name*, returning a display object. If unable to open a connection, `#f` is returned. *Display-name* is normally a string, which is an X display name in the usual form; however, `#f` is also allowed, meaning to use the value of the unix environment variable `DISPLAY`.

`x-graphics/close-display` *display* [procedure]
Closes *display*; after calling this procedure, it is an error to use *display* for any purpose. Any windows that were previously opened on *display* are destroyed and their resources returned to the operating system.

`x-close-all-displays` [procedure]
Closes all open connections to X displays. Equivalent to calling `x-close-display` on all open displays.

`x-geometry-string` *x y width height* [procedure]
This procedure creates and returns a standard X geometry string from the given arguments. *X* and *y* must be either exact integers or `#f`, while *width* and *height* must be either exact non-negative integers or `#f`. Usually either *x* and *y* are both specified or both `#f`; similarly for *width* and *height*. If only one of the elements of such a pair is specified, it is ignored.

Examples:

```
(x-geometry-string #f #f 100 200) ⇒ "100x200"
(x-geometry-string 2 -3 100 200) ⇒ "100x200+2-3"
(x-geometry-string 2 -3 #f #f) ⇒ "+2-3"
```

Note that the *x* and *y* arguments cannot distinguish between `+0` and `-0`, even though these have different meanings in X. If either of those arguments is `0`, it means `+0` in X terminology. If you need to distinguish these two cases you must create your own geometry string using Scheme's string and number primitives.

17.9.3 Custom Operations on X Graphics Devices

Custom operations are invoked using the procedure `graphics-operation`. For example,

```
(graphics-operation device 'set-foreground-color "blue")
```

`set-background-color` *color-name* [operation on `x-graphics-device`]
`set-foreground-color` *color-name* [operation on `x-graphics-device`]
`set-border-color` *color-name* [operation on `x-graphics-device`]
`set-mouse-color` *color-name* [operation on `x-graphics-device`]
These operations change the colors associated with a window. *Color-name* must be a string, which is the X server's name for the desired color. `set-border-color` and `set-mouse-color` immediately change the border and mouse-cursor colors. `set-background-color` and `set-foreground-color` change the colors to be used when drawing, but have no effect on anything drawn prior to their invocation. Because changing the background color affects the entire window, we recommend calling `graphics-clear` on the window's device afterwards. Color names include both mnemonic names, like `"red"`, and intensity names specified in the `"#rrggbb"` notation.

`draw-arc` *x y radius-x radius-y angle-start angle-sweep fill?* [operation on `x-graphics-device`]
Operation `draw-arc` draws or fills an arc. An arc is a segment of a circle, which may have been stretched along the x- or y- axis to form an ellipse.

The parameters *x*, *y*, *radius-x* and *radius-y* describe the circle and *angle-start* and *angle-sweep* choose which part of the circle is drawn. The arc is drawn on the graphics device with the center of the circle at the virtual coordinates given by *x* and *y*. *radius-x* and *radius-y* determine the size of the circle in virtual coordinate units.

The parameter *angle-start* determines where the arc starts. It is measured in degrees in an anti-clockwise direction, starting at 3 o'clock. *angle-sweep* determines how much of the circle is drawn. It too is measured anti-clockwise in degrees. A negative value means the measurement is in a clockwise direction.

Note that the angles are determined on a unit circle before it is stretched into an ellipse, so the actual angles that you will see on the computer screen depends on all of: *radius-x* and *radius-y*, the window size, and the virtual coordinates.

If *fill?* is `#f` then just the segment of the circle is drawn, otherwise the arc is filled in a pie-slice fashion.

This draws a quarter circle pie slice, standing on its point, with point at virtual coordinates (3,5):

```
(graphics-operation g 'draw-arc 3 5 .5 .5 45 90 #t)
```

`draw-circle` *x y radius* [operation on `x-graphics-device`]
`fill-circle` *x y radius* [operation on `x-graphics-device`]
These operations draw a circle (outline) or a filled circle (solid) at on the graphics device at the virtual coordinates given by *x* and *y*. These operations could be implemented trivially interms of the `draw-arc` operation.

`set-border-width` *width* [operation on `x-graphics-device`]
`set-internal-border-width` *width* [operation on `x-graphics-device`]
These operations change the external and internal border widths of a window. *Width* must be an exact non-negative integer, specified in pixels. The change takes place immediately. Note that changing the internal border width can cause displayed graphics to be garbled; we recommend calling `graphics-clear` on the window's device after doing so.

`set-font` *font-name* [operation on `x-graphics-device`]
Changes the font used when drawing text in a window. *Font-name* must be a string that is a font name known to the X server. This operation does not affect text drawn prior to its invocation.

`set-mouse-shape` *shape-number* [operation on `x-graphics-device`]
Changes the shape of the mouse cursor. *Shape-number* is an exact non-negative integer that is used as an index into the mouse-shape font; when multiplied by 2 this number corresponds to an index in the file `/usr/include/X11/cursorfont.h`.

`map-window` [operation on `x-graphics-device`]
`withdraw-window` [operation on `x-graphics-device`]
These operations control the mapping of windows. They correspond directly to Xlib's `XMapWindow` and `XWithdrawWindow`.

`resize-window` *width height* [operation on `x-graphics-device`]
Changes the size of a window. *Width* and *height* must be exact non-negative integers. The operation corresponds directly to Xlib's `XResizeWindow`.

This operation resets the virtual coordinate system and the clip rectangle.

`move-window` *x y* [operation on `x-graphics-device`]
Changes the position of a window on the display. *X* and *y* must be exact integers. The operation corresponds directly to Xlib's `XMoveWindow`. Note that the coordinates *x* and *y* do not take the external border into account, and therefore will not position the window as you might like. The only reliable way to position a window is to ask a window manager to do it for you.

`get-default` *resource property* [operation on `x-graphics-device`]
This operation corresponds directly to Xlib's `XGetDefault`. *Resource* and *property* must be strings. The operation returns the character string corresponding to the association of *resource* and *property*; if no such association exists, `#f` is returned.

`copy-area` *source-x-left source-y-top width height destination-x-left destination-y-top* [operation on `x-graphics-device`]
This operation copies the contents of the rectangle specified by *source-x-left*, *source-y-top*, *width*, and *height* to the rectangle of the same dimensions at *destination-x-left* and *destination-y-top*.

`font-structure` *font-name* [operation on `x-graphics-device`]
Returns a Scheme equivalent of the X font structure for the font named *font-name*. If the string *font-name* does not name a font known to the X server, or names a 16-bit font, `#f` is returned.

`x-font-structure/name` *font-structure* [procedure]
`x-font-structure/direction` *font-structure* [procedure]
`x-font-structure/all-chars-exist` *font-structure* [procedure]
`x-font-structure/default-char` *font-structure* [procedure]
`x-font-structure/min-bounds` *font-structure* [procedure]
`x-font-structure/max-bounds` *font-structure* [procedure]
`x-font-structure/start-index` *font-structure* [procedure]
`x-font-structure/character-bounds` *font-structure* [procedure]
`x-font-structure/max-ascent` *font-structure* [procedure]
`x-font-structure/max-descent` *font-structure* [procedure]
These procedures extract the components of the font description structure returned by the X graphics operation `font-structure`. A more complete description of these components appears in documentation of the `XLoadQueryFont` Xlib call. `start-index` is the index of the first character available in the font. The `min-bounds` and `max-bounds` components are structures of type `x-character-bounds`, and the `character-bounds` component is a vector of the same type.

`x-character-bounds/lbearing` *character-bounds* [procedure]
`x-character-bounds/rbearing` *character-bounds* [procedure]
`x-character-bounds/width` *character-bounds* [procedure]
`x-character-bounds/ascent` *character-bounds* [procedure]

`x-character-bounds/descent` *character-bounds* [procedure]

These procedures extract components of objects of type `x-character-bounds`. A more complete description of them appears in documentation of the `XLoadQueryFont` Xlib call.

18 Standards Support

18.1 Revised[7] Report on the Algorithmic Language Scheme

Description The core standard for the Scheme language.

URL https://small.r7rs.org/attachment/r7rs.pdf

Support Fully supported, with exceptions:

- Multiple values aren't supported correctly. Instead `values` returns a special record containing the values, and `call-with-values` takes that record apart and passes the values to the receiver. For many uses of multiple values this is adequate.

Libraries `(scheme base)` `(scheme case-lambda)` `(scheme char)` `(scheme complex)` `(scheme cxr)` `(scheme eval)` `(scheme file)` `(scheme inexact)` `(scheme lazy)` `(scheme load)` `(scheme process-context)` `(scheme r5rs)` `(scheme read)` `(scheme repl)` `(scheme time)` `(scheme write)`

Global All names are bound in the MIT/GNU Scheme global environment.

`*`	`+`	`-`
`...`	`/`	`<`
`<=`	`=`	`=>`
`>`	`>=`	`_`
`abs`	`acos`	`and`
`angle`	`append`	`apply`
`asin`	`assoc`	`assq`
`assv`	`atan`	`begin`
`binary-port?`	`boolean=?`	`boolean?`
`bytevector`	`bytevector-append`	`bytevector-copy`
`bytevector-copy!`	`bytevector-length`	`bytevector-u8-ref`
`bytevector-u8-set!`	`bytevector?`	`caaaar`
`caaadr`	`caaar`	`caadar`
`caaddr`	`caadr`	`caar`
`cadaar`	`cadadr`	`cadar`
`caddar`	`cadddr`	`caddr`
`cadr`	`call-with-current-continuation`	`call-with-input-file`
`call-with-output-file`	`call-with-port`	`call-with-values`
`call/cc`	`car`	`case`
`case-lambda`	`cdaaar`	`cdaadr`
`cdaar`	`cdadar`	`cdaddr`
`cdadr`	`cdar`	`cddaar`
`cddadr`	`cddar`	`cdddar`

```
cddddr                  cdddr                   cddr
cdr                     ceiling                 char->integer
char-alphabetic?        char-ci<=?              char-ci<?
char-ci=?               char-ci>=?              char-ci>?
char-downcase           char-foldcase           char-lower-case?
char-numeric?           char-ready?             char-upcase
char-upper-case?        char-whitespace?        char<=?
char<?                  char=?                  char>=?
char>?                  char?                   close-input-port
close-output-port       close-port              command-line
complex?                cond                    cond-expand
cons                    cos                     current-error-port
current-input-port      current-jiffy           current-output-port
current-second          define                  define-record-type
define-syntax           define-values           delay
delay-force             delete-file             denominator
digit-value             display                 do
dynamic-wind            else                    emergency-exit
environment             eof-object              eof-object?
eq?                     equal?                  eqv?
error                   error-object-irritants  error-object-message
error-object?           eval                    even?
exact                   exact-integer-sqrt      exact-integer?
exact?                  exit                    exp
expt                    features                file-error?
file-exists?            finite?                 floor
floor-quotient          floor-remainder         floor/
flush-output-port       for-each                force
gcd                     get-environment-        get-environment-
                        variable                variables
get-output-bytevector   get-output-string       guard
if                      imag-part               include
include-ci              inexact                 inexact?
infinite?               input-port-open?        input-port?
integer->char           integer?                interaction-
                                                environment
jiffies-per-second      lambda                  lcm
length                  let                     let*
let*-values             let-syntax              let-values
letrec                  letrec*                 letrec-syntax
list                    list->string            list->vector
list-copy               list-ref                list-set!
list-tail               list?                   load
log                     magnitude               make-bytevector
make-list               make-parameter          make-polar
make-promise            make-rectangular        make-string
```

```
make-vector                 map                         max
member                      memq                        memv
min                         modulo                      nan?
negative?                   newline                     not
null?                       number->string              number?
numerator                   odd?                        open-binary-input-file
open-binary-output-file     open-input-bytevector       open-input-file
open-input-string           open-output-bytevector      open-output-file
open-output-string          or                          output-port-open?
output-port?                pair?                       parameterize
peek-char                   peek-u8                     port?
positive?                   procedure?                  promise?
quasiquote                  quote                       quotient
raise                       raise-continuable           rational?
rationalize                 read                        read-bytevector
read-bytevector!            read-char                   read-error?
read-line                   read-string                 read-u8
real-part                   real?                       remainder
reverse                     round                       set!
set-car!                    set-cdr!                    sin
sqrt                        square                      string
string->list                string->number              string->symbol
string->utf8                string->vector              string-append
string-ci<=?                string-ci<?                 string-ci=?
string-ci>=?                string-ci>?                 string-copy
string-copy!                string-downcase             string-fill!
string-foldcase             string-for-each             string-length
string-map                  string-ref                  string-set!
string-upcase               string<=?                   string<?
string=?                    string>=?                   string>?
string?                     substring                   symbol->string
symbol=?                    symbol?                     syntax-error
syntax-rules                tan                         textual-port?
truncate                    truncate-quotient           truncate-remainder
truncate/                   u8-ready?                   unless
unquote                     unquote-splicing            utf8->string
values                      vector                      vector->list
vector->string              vector-append               vector-copy
vector-copy!                vector-fill!                vector-for-each
vector-length               vector-map                  vector-ref
vector-set!                 vector?                     when
with-exception-handler      with-input-from-file        with-output-to-file
write                       write-bytevector            write-char
write-shared                write-simple                write-string
```

18.2 SRFI 1: List Library

Description An extensive set of procedures for working with lists that is a superset of the list procedures defined by R7RS.

URL https://srfi.schemers.org/srfi-1/srfi-1.html

Support Fully supported.

Libraries `(srfi 1)`

Global All names are bound in the MIT/GNU Scheme global environment.

`alist-cons`	`alist-copy`	`alist-delete`
`alist-delete!`	`any`	`append`
`append!`	`append-map`	`append-map!`
`append-reverse`	`append-reverse!`	`assoc`
`assq`	`assv`	`break`
`break!`	`caaaar`	`caaadr`
`caaar`	`caadar`	`caaddr`
`caadr`	`caar`	`cadaar`
`cadadr`	`cadar`	`caddar`
`cadddr`	`caddr`	`cadr`
`car`	`car+cdr`	`cdaaar`
`cdaadr`	`cdaar`	`cdadar`
`cdaddr`	`cdadr`	`cdar`
`cddaar`	`cddadr`	`cddar`
`cdddar`	`cddddr`	`cdddr`
`cddr`	`cdr`	`circular-list`
`circular-list?`	`concatenate`	`concatenate!`
`cons`	`cons*`	`count`
`delete`	`delete!`	`delete-duplicates`
`delete-duplicates!`	`dotted-list?`	`drop`
`drop-right`	`drop-right!`	`drop-while`
`eighth`	`every`	`fifth`
`filter`	`filter!`	`filter-map`
`find`	`find-tail`	`first`
`fold`	`fold-right`	`for-each`
`fourth`	`iota`	`last`
`last-pair`	`length`	`length+`
`list`	`list-copy`	`list-index`
`list-ref`	`list-tabulate`	`list=`
`lset-adjoin`	`lset-diff+intersection`	`lset-diff+intersection!`
`lset-difference`	`lset-difference!`	`lset-intersection`
`lset-intersection!`	`lset-union`	`lset-union!`
`lset-xor`	`lset-xor!`	`lset<=`

```
lset=                   make-list               map
map!                    map-in-order            member
memq                    memv                    ninth
not-pair?               null-list?              null?
pair-fold               pair-fold-right         pair-for-each
pair?                   partition               partition!
proper-list?            reduce                  reduce-right
remove                  remove!                 reverse
reverse!                second                  set-car!
set-cdr!                seventh                 sixth
span                    span!                   split-at
split-at!               take                    take!
take-right              take-while              take-while!
tenth                   third                   unfold
unfold-right            unzip1                  unzip2
unzip3                  unzip4                  unzip5
xcons                   zip
```

18.3 SRFI 2: `and-let*`

Description	A macro combining features of `let*` and `and`.
URL	https://srfi.schemers.org/srfi-2/srfi-2.html
Support	Fully supported.
Libraries	`(srfi 2)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`and-let*`

18.4 SRFI 8: `receive`

Description	Early syntax for binding multiple values. R7RS provides the more flexible `let-values`, which should be preferred.
URL	https://srfi.schemers.org/srfi-8/srfi-8.html
Support	Fully supported.
Libraries	`(srfi 8)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`receive`

18.5 SRFI 9: Record Types

Description	A basic implementation of record types. Superseded by R7RS.
URL	https://srfi.schemers.org/srfi-9/srfi-9.html
Support	The MIT/GNU Scheme implementation is fully compatible.
Libraries	`(srfi 9)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`define-record-type`

18.6 SRFI 14: Character-set Library

Description	An extensive set of definitions for working with sets of characters.
URL	https://srfi.schemers.org/srfi-14/srfi-14.html
Support	Fully supported.
Libraries	`(srfi 14)`
Global	Most names are bound in the MIT/GNU Scheme global environment. Exceptions are the linear-update procedures whose names end in `!`.

`->char-set`
`char-set`
`char-set->list`
`char-set->string`
`char-set-adjoin`
`char-set-adjoin!`
`char-set-any`
`char-set-complement`
`char-set-complement!`
`char-set-contains?`
`char-set-copy`
`char-set-count`
`char-set-cursor`
`char-set-cursor-next`
`char-set-delete`
`char-set-delete!`
`char-set-diff+intersection`
`char-set-diff+intersection!`
`char-set-difference`
`char-set-difference!`
`char-set-every`
`char-set-filter`
`char-set-filter!`
`char-set-fold`
`char-set-for-each`
`char-set-hash`
`char-set-intersection`
`char-set-intersection!`
`char-set-map`
`char-set-ref`
`char-set-size`
`char-set-unfold`
`char-set-unfold!`
`char-set-union`
`char-set-union!`
`char-set-xor`
`char-set-xor!`
`char-set:ascii`

```
char-set:blank                  char-set:digit
char-set:empty                  char-set:full
char-set:graphic                char-set:hex-digit
char-set:iso-control            char-set:letter
char-set:letter+digit           char-set:lower-case
char-set:printing               char-set:punctuation
char-set:symbol                 char-set:title-case
char-set:upper-case             char-set:whitespace
char-set<=                      char-set=
char-set?                       end-of-char-set?
list->char-set                  list->char-set!
string->char-set                string->char-set!
ucs-range->char-set             ucs-range->char-set!
```

18.7 SRFI 23: Error Reporting Mechanism

Description A simple implementation of `error`. Superseded by R7RS.

URL https://srfi.schemers.org/srfi-23/srfi-23.html

Support Fully supported.

Libraries `(srfi 23)`

Global All names are bound in the MIT/GNU Scheme global environment.

`error`

18.8 SRFI 27: Sources of Random Bits

Description Procedures to obtain pseudo-random numbers in various formats.

URL https://srfi.schemers.org/srfi-27/srfi-27.html

Support Fully supported.

Libraries `(srfi 27)`

Global All names are bound in the MIT/GNU Scheme global environment.

```
default-random-source           make-random-source
random-integer                  random-real
random-source-make-integers     random-source-make-reals
random-source-make-reals        random-source-pseudo-randomize!
random-source-randomize!        random-source-state-ref
random-source-state-set!        random-source?
```

18.9 SRFI 39: Parameter Objects

Description	A mechanism for dynamic binding. Superseded by R7RS.
URL	https://srfi.schemers.org/srfi-39/srfi-39.html
Support	Fully supported.
Libraries	`(srfi 39)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`make-parameter` `parameterize`

18.10 SRFI 69: Basic Hash Tables

Description	A standard interface for hash tables. Superseded by SRFI 125.
URL	https://srfi.schemers.org/srfi-69/srfi-69.html
Support	Fully supported.
Libraries	`(srfi 69)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`alist->hash-table` `hash`
`hash-by-identity` `hash-table->alist`
`hash-table-copy` `hash-table-delete!`
`hash-table-equivalence-function` `hash-table-exists?`
`hash-table-fold` `hash-table-hash-function`
`hash-table-keys` `hash-table-merge!`
`hash-table-ref` `hash-table-ref/default`
`hash-table-set!` `hash-table-size`
`hash-table-update!` `hash-table-update!/default`
`hash-table-values` `hash-table-walk`
`hash-table?` `make-hash-table`
`string-ci-hash` `string-hash`

18.11 SRFI 112: Environment Inquiry

Description	Provides human-readable information at run time about the hardware and software configuration on which a Scheme program is being executed.
URL	https://srfi.schemers.org/srfi-112/srfi-112.html
Support	Fully supported.
Libraries	`(srfi 112)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`cpu-architecture`
`implementation-name`
`implementation-version`
`machine-name`
`os-name`
`os-version`

18.12 SRFI 115: Scheme Regular Expressions

Description	An implementation of regular expressions using Scheme syntax.
URL	https://srfi.schemers.org/srfi-115/srfi-115.html
Support	Fully supported.
Libraries	`(srfi 115)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`char-set->sre`
`regexp`
`regexp-extract`
`regexp-fold`
`regexp-match->list`
`regexp-match-count`
`regexp-match-submatch`
`regexp-match-submatch-end`
`regexp-match-submatch-start`
`regexp-match?`
`regexp-matches`
`regexp-matches?`
`regexp-partition`
`regexp-replace`
`regexp-replace-all`
`regexp-search`
`regexp-split`
`regexp?`
`rx`
`valid-sre?`

18.13 SRFI 124: Ephemerons

Description Support for ephemerons, a weak-pointer mechanism.

URL https://srfi.schemers.org/srfi-124/srfi-124.html

Support Fully supported.

Libraries `(srfi 124)`

Global All names are bound in the MIT/GNU Scheme global environment.

`ephemeron-broken?` `ephemeron-datum` `ephemeron-key`
`ephemeron?` `make-ephemeron` `reference-barrier`

18.14 SRFI 125: Intermediate Hash Tables

Description A comprehensive set of procedures for hash tables. Supersedes SRFI 69.

URL https://srfi.schemers.org/srfi-125/srfi-125.html

Support Fully supported.

Libraries `(srfi 125)`

Global All names are bound in the MIT/GNU Scheme global environment.

`alist->hash-table`
`hash-by-identity`
`hash-table->alist`
`hash-table-contains?`
`hash-table-count`
`hash-table-difference!`
`hash-table-empty?`
`hash-table-equivalence-function`
`hash-table-find`
`hash-table-for-each`
`hash-table-intern!`
`hash-table-keys`
`hash-table-map!`
`hash-table-merge!`
`hash-table-pop!`
`hash-table-ref`
`hash-table-set!`
`hash-table-unfold`
`hash-table-update!`
`hash-table-values`
`hash`
`hash-table`
`hash-table-clear!`
`hash-table-copy`
`hash-table-delete!`
`hash-table-empty-copy`
`hash-table-entries`
`hash-table-exists?`
`hash-table-fold`
`hash-table-hash-function`
`hash-table-intersection!`
`hash-table-map`
`hash-table-map->list`
`hash-table-mutable?`
`hash-table-prune!`
`hash-table-ref/default`
`hash-table-size`
`hash-table-union!`
`hash-table-update!/default`
`hash-table-walk`

```
hash-table-xor!
hash-table?
string-ci-hash
hash-table=?
make-hash-table
string-hash
```

18.15 SRFI 128: Comparators (reduced)

Description	An implementation of comparators, which encapsulate type, equality, ordering, and hashing.
URL	https://srfi.schemers.org/srfi-128/srfi-128.html
Support	Fully supported.
Libraries	`(srfi 128)`
Global	All names are bound in the MIT/GNU Scheme global environment.

```
<=?
=?
>?
char-ci-hash
comparator-check-type
comparator-hash
comparator-hashable?
comparator-ordered?
comparator-register-default!
comparator-type-test-predicate
default-hash
hash-salt
make-default-comparator
make-equal-comparator
make-list-comparator
make-vector-comparator
string-ci-hash
symbol-hash
<?
>=?
boolean-hash
char-hash
comparator-equality-predicate
comparator-hash-function
comparator-if<=>
comparator-ordering-predicate
comparator-test-type
comparator?
hash-bound
make-comparator
make-eq-comparator
make-eqv-comparator
make-pair-comparator
number-hash
string-hash
```

18.16 SRFI 129: Titlecase procedures

Description	An implementation procedures for title case.
URL	https://srfi.schemers.org/srfi-129/srfi-129.html
Support	Fully supported.
Libraries	`(srfi 129)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`char-title-case?` `char-titlecase` `string-titlecase`

18.17 SRFI 131: ERRR5RS Record Syntax (reduced)

Description	An enhanced record syntax supporting abbreviations and single inheritance.
URL	https://srfi.schemers.org/srfi-131/srfi-131.html
Support	The MIT/GNU Scheme implementation is fully compatible.
Libraries	`(srfi 131)`
Global	All names are bound in the MIT/GNU Scheme global environment.

`define-record-type`

18.18 SRFI 133: Vector Library (R7RS-compatible)

Description	A set of procedures for working with vectors.
URL	https://srfi.schemers.org/srfi-133/srfi-133.html
Support	Fully supported.
Libraries	`(srfi 133)`
Global	No bindings in the MIT/GNU Scheme global environment. Some of the same names are there, but they aren't the same implementations.

`reverse-list->vector`
`reverse-vector->list`
`vector-any`
`vector-append-subvectors`
`vector-cumulate`
`vector-empty?`
`vector-every`
`vector-fold`
`vector-fold-right`
`vector-index`
`vector-index-right`
`vector-map!`

```
vector-partition
vector-reverse-copy
vector-skip-right
vector-unfold
vector-unfold-right
vector=
vector-concatenate
vector-reverse-copy!
vector-reverse!
vector-skip
vector-swap!
vector-unfold!
vector-unfold-right!
vector-binary-search
vector-count
```

18.19 SRFI 140: Immutable Strings

Description	A comprehensive library for using immutable strings.
URL	https://srfi.schemers.org/srfi-140/srfi-140.html
Support	Partially supported: only immutable strings are implemented.
Libraries	`(srfi 140)`
Global	Some bindings in the MIT/GNU Scheme global environment; see below.

Names that are bound in the MIT/GNU Scheme global environment:

```
istring?
string
string->utf16
string->utf16le
string->vector
string-ci<=?
string-ci=?
string-ci>?
string-downcase
string-fold-right
string-for-each
string-map
string-ref
string-upcase
string<?
string>=?
string?
utf16->string
utf16le->string
vector->string
list->string
string->list
string->utf16be
string->utf8
string-append
string-ci<?
string-ci>=?
string-concatenate
string-fold
string-foldcase
string-length
string-null?
string-titlecase
string<=?
string=?
string>?
substring
utf16be->string
utf8->string
```

Names that are not bound in the MIT/GNU Scheme global environment:

```
reverse-list->string
string-concatenate-reverse
string-contains-right
string-any
string-contains
string-count
```

```
string-drop                 string-drop-right
string-every                string-filter
string-for-each-index       string-index
string-index-right          string-join
string-map-index            string-pad
string-pad-right            string-prefix-length
string-prefix?              string-remove
string-repeat               string-replace
string-skip                 string-skip-right
string-split                string-suffix-length
string-suffix?              string-tabulate
string-take                 string-take-right
string-trim                 string-trim-both
string-trim-right           string-unfold
string-unfold-right         xsubstring
```

18.20 SRFI 143: Fixnums

Description	Definitions of fixnum-specific procedures.
URL	https://srfi.schemers.org/srfi-143/srfi-143.html
Support	Fully supported with efficient compilation.
Libraries	`(srfi 143)`
Global	All names are bound in the MIT/GNU Scheme global environment.

```
fixnum?                     fx*
fx*/carry                   fx+
fx+/carry                   fx-
fx-/carry                   fx-greatest
fx-least                    fx-width
fx<=?                       fx<?
fx=?                        fx>=?
fx>?                        fxabs
fxand                       fxarithmetic-shift
fxarithmetic-shift-left     fxarithmetic-shift-right
fxbit-count                 fxbit-field
fxbit-field-reverse         fxbit-field-rotate
fxbit-set?                  fxcopy-bit
fxeven?                     fxfirst-set-bit
fxif                        fxior
fxlength                    fxmax
fxmin                       fxneg
fxnegative?                 fxnot
```

```
fxodd?                                  fxpositive?
fxquotient                              fxremainder
fxsqrt                                  fxsquare
fxxor                                   fxzero?
```

18.21 SRFI 158: Generators and Accumulators

Description Efficient sources and sinks of objects.

URL https://srfi.schemers.org/srfi-158/srfi-158.html

Support Fully supported.

Libraries `(srfi 158)`

Global All names are bound in the MIT/GNU Scheme global environment.

```
bytevector->generator                   bytevector-accumulator
bytevector-accumulator!                 circular-generator
count-accumulator                       gappend
gcombine                                gcons*
gdelete                                 gdelete-neighbor-dups
gdrop                                   gdrop-while
generator                               generator->list
generator->reverse-list                 generator->string
generator->vector                       generator->vector!
generator-any                           generator-count
generator-every                         generator-find
generator-fold                          generator-for-each
generator-map->list                     generator-unfold
gfilter                                 gflatten
ggroup                                  gindex
gmap                                    gmerge
gremove                                 gselect
gstate-filter                           gtake
gtake-while                             list->generator
list-accumulator                        make-accumulator
make-coroutine-generator                make-for-each-generator
make-iota-generator                     make-range-generator
make-unfold-generator                   product-accumulator
reverse-list-accumulator                reverse-vector->generator
reverse-vector-accumulator              string->generator
string-accumulator                      sum-accumulator
vector->generator                       vector-accumulator
vector-accumulator!
```

18.22 SRFI 162: Comparators sublibrary

Description	Additional useful comparator definitions.
URL	https://srfi.schemers.org/srfi-162/srfi-162.html
Support	Fully supported.
Libraries	`(srfi 128)` [*not* `(srfi 162)`]
Global	All names are bound in the MIT/GNU Scheme global environment.

`boolean-comparator`
`char-ci-comparator`
`char-comparator`
`comparator-max`
`comparator-max-in-list`
`comparator-min`
`comparator-min-in-list`
`default-comparator`
`eq-comparator`
`equal-comparator`
`eqv-comparator`
`list-comparator`
`pair-comparator`
`real-comparator`
`string-ci-comparator`
`string-comparator`
`vector-comparator`

18.23 SRFI 219: Define higher-order lambda

Description	Higher-order lambda syntax for DEFINE
URL	https://srfi.schemers.org/srfi-219/srfi-219.html
Support	Fully supported.
Libraries	No libraries.
Global	All names are bound in the MIT/GNU Scheme global environment.

`define`

Appendix A GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

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.

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 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 Warrany 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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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.

A.1 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 1.2
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.

Appendix B Binding Index

"

" 117

#

..... 66
#!optional 12, 16
#!rest 12, 16
#(..... 157
#\ 109
#\alarm 109
#\backspace 109
#\delete 109
#\escape 109
#\newline 109
#\null 109
#\return 109
#\space 109, 127
#\tab 109
#b 66
#d 66
#e 66
#f 8, 26, 167
#i 66
#o 66
#t 8, 26, 167
#x 66

'

' 24, 139

(

(..... 139
() 139

)

) 139

*

* 14, 68, 134, 261, 263, 345
** 135, 136
**? 135
*? 134
default-pathname-defaults 282
*matcher 259
*parser 262
parser-canonicalize-symbols? 238
parser-radix 238
random-state 105
unparse-with-maximum-readability? 241
unparser-list-breadth-limit 241
unparser-list-depth-limit 241
unparser-radix 241
unparser-string-length-limit 241

+

+ 14, 64, 68, 135, 261, 263, 345
+? 135
+inf.0 90
+nan.0 90
+snan.1 90

,

, ... 25, 139
,@ 25, 139

—

- 68, 345
->char-set 350
->namestring 276
->pathname 276, 285
->truename 286
-1+ 68
-inf.0 90
-nan.0 90
-snan.1 90

.

. 139
... . 40, 345

/

/ 65, 68, 345

<

< 67, 345
<= 67, 345
<=? 355
<? 355
<xml-!attlist> 272
<xml-!element> 271
<xml-!entity> 272
<xml-!notation> 272
<xml-declaration> 271
<xml-document> 271
<xml-dtd> 271
<xml-element> 271
<xml-external-id> 272
<xml-parameter-!entity> 272
<xml-processing-instructions> 271
<xml-unparsed-!entity> 272

=

= 57, 66, 67, 124, 345
=> 27, 345
=? 355

>

> 67, 345
>= 67, 345
>=? 355
>? 355

?

? 134, 261, 263
?? 134

_

_ 345

`

` 25, 139

\

\" 117
\\ 117
\| 117
\a 117
\b 117
\n 117
\r 117
\t 117
\x 117

1

1+ 68
1d-table/alist 184
1d-table/get 184
1d-table/lookup 184
1d-table/put! 184
1d-table/remove! 184
1d-table? 184

2

2d-get 185
2d-get-alist-x 185
2d-get-alist-y 185
2d-put! 184
2d-remove! 184

8

8-bit-char-set? 116

A

abort 315, 318
abs 68, 345
access 24
access-condition 321, 323
acos 74, 345
acos/pi 80
aexsec 76
aexsec/pi 80
alist->amap 209
alist->hash-table 187, 352, 354
alist->rb-tree 200
alist->wt-tree 203
alist-cons 347
alist-copy 183, 347
alist-delete 347
alist-delete! 347
alist? 182
allocate-host-address 307
alt 136, 261, 263
amap->alist 213
amap-args 211
amap-clean! 211
amap-clear! 211
amap-comparator 210
amap-contains? 210
amap-copy 213
amap-count 213
amap-delete! 211
amap-difference! 213
amap-empty-copy 213
amap-empty? 210
amap-entries 212
amap-find 213
amap-fold 212
amap-for-each 212

amap-implementation-name ... 211
amap-implementation-names ... 209
amap-implementation-supported-args ... 210
amap-implementation-supports-args? ... 210
amap-implementation-supports-comparator? ... 210
amap-intern! ... 211
amap-intersection! ... 213
amap-keys ... 212
amap-map ... 212
amap-map! ... 212
amap-map->list ... 212
amap-mutable? ... 210
amap-pop! ... 211
amap-prune! ... 212
amap-ref ... 210
amap-ref/default ... 210
amap-set! ... 211
amap-size ... 212
amap-unfold ... 209
amap-union! ... 213
amap-update! ... 211
amap-update!/default ... 211
amap-values ... 212
amap-xor! ... 213
amap=? ... 210
amap? ... 210
and ... 28, 167, 345
and-let* ... 54, 349
angle ... 74, 81, 345
any ... 151, 347
any-char ... 132
append ... 146, 345, 347
append! ... 146, 347
append-map ... 150, 347
append-map! ... 151, 347
append-map* ... 150
append-map*! ... 151
append-reverse ... 347
append-reverse! ... 347
apply ... 215, 345
apply-hook-extra ... 221
apply-hook-procedure ... 221
apply-hook? ... 221
arithmetic-shift ... 85
asin ... 74, 345
asin/pi ... 80
assoc ... 182, 345, 347
association-procedure ... 182
assq ... 182, 345, 347
assv ... 182, 345, 347
atan ... 74, 95, 345
atan/pi ... 80
atan2/pi ... 80
aversin ... 76
aversin/pi ... 80

B

beep ... 240
begin ... 28, 345
binary-port? ... 228, 345
bind-cell-contents! ... 172
bind-condition-handler ... 310, 311, 313, 314
bind-default-condition-handler ... 311, 313, 314
bit ... 87
bit-antimask ... 86
bit-clear? ... 86
bit-count ... 86
bit-mask ... 85
bit-set? ... 86
bit-string->signed-integer ... 166
bit-string->unsigned-integer ... 166
bit-string-allocate ... 163
bit-string-and ... 165
bit-string-and! ... 165
bit-string-andc ... 165
bit-string-andc! ... 165
bit-string-append ... 164
bit-string-clear! ... 164
bit-string-copy ... 163
bit-string-fill! ... 165
bit-string-length ... 164
bit-string-move! ... 166
bit-string-movec! ... 165
bit-string-not ... 165
bit-string-or ... 165
bit-string-or! ... 165
bit-string-ref ... 164
bit-string-set! ... 164
bit-string-xor ... 165
bit-string-xor! ... 165
bit-string-zero? ... 165
bit-string=? ... 165
bit-string? ... 164
bit-substring ... 164
bit-substring-find-next-set-bit ... 164
bit-substring-move-right! ... 166
bitless-char? ... 113
bits ... 87
bitwise-and ... 85
bitwise-andc1 ... 85
bitwise-andc2 ... 85
bitwise-eqv ... 85
bitwise-ior ... 85
bitwise-nand ... 85
bitwise-nor ... 85
bitwise-not ... 85
bitwise-orc1 ... 85
bitwise-orc2 ... 85
bitwise-xor ... 85
boolean-comparator ... 360
boolean-hash ... 355
boolean/and ... 168
boolean/or ... 168
boolean=? ... 345

boolean? ... 167, 345
both ... 128
bound-restart ... 319
bound-restarts ... 319
bracketed-print-method ... 246
break ... 347
break! ... 347
break-on-signals ... 311, 313, 314
buffered-input-chars on
  textual input port ... 252
buffered-output-chars on
  textual output port ... 253
bytevector ... 345
bytevector->generator ... 359
bytevector-accumulator ... 359
bytevector-accumulator! ... 359
bytevector-append ... 345
bytevector-copy ... 345
bytevector-copy! ... 345
bytevector-length ... 345
bytevector-u8-ref ... 345
bytevector-u8-set! ... 345
bytevector? ... 345

C

caaaar ... 141, 345, 347
caaadr ... 141, 345, 347
caaar ... 141, 345, 347
caadar ... 141, 345, 347
caaddr ... 141, 345, 347
caadr ... 141, 345, 347
caar ... 141, 345, 347
cadaar ... 141, 345, 347
cadadr ... 141, 345, 347
cadar ... 141, 345, 347
caddar ... 141, 345, 347
cadddr ... 141, 345, 347
caddr ... 141, 345, 347
cadr ... 141, 345, 347
call-with-binary-input-file ... 230
call-with-binary-output-file ... 230
call-with-current-continuation ... 218, 345
call-with-input-file ... 229, 345
call-with-output-bytevector ... 233
call-with-output-file ... 229, 345
call-with-output-string ... 231
call-with-port ... 227, 229, 230, 345
call-with-temporary-file-pathname ... 286
call-with-truncated-output-port ... 227, 232
call-with-truncated-output-string ... 232
call-with-values ... 220, 345
call/cc ... 345
canonical-host-name ... 307
capture-syntactic-environment ... 46
car ... 9, 140, 177, 345, 347
car+cdr ... 141, 347
case ... 27, 29, 345
case-lambda ... 345
cd ... 285
cdaaar ... 141, 345, 347
cdaadr ... 141, 345, 347
cdaar ... 141, 345, 347
cdadar ... 141, 345, 347
cdaddr ... 141, 345, 347
cdadr ... 141, 345, 347
cdar ... 141, 345, 347
cddaar ... 141, 345, 347
cddadr ... 141, 345, 347
cddar ... 141, 345, 347
cdddar ... 141, 345, 347
cddddr ... 141, 345, 347
cdddr ... 141, 345, 347
cddr ... 141, 345, 347
cdr ... 6, 140, 177, 345, 347
ceiling ... 73, 345
ceiling->exact ... 73
ceiling-quotient ... 71
ceiling-remainder ... 71
ceiling/ ... 69
cell-contents ... 172
cell? ... 172
char ... 259
char->bitless-char ... 113
char->digit ... 112
char->integer ... 111, 345
char->name ... 110
char-alphabetic? ... 110, 345
char-alphanumeric? ... 111
char-bits ... 113
char-bits-limit ... 113
char-ci ... 132, 259
char-ci-comparator ... 360
char-ci-hash ... 355
char-ci-predicate ... 113
char-ci<=? ... 110, 345
char-ci<? ... 110, 345
char-ci=? ... 110, 345
char-ci>=? ... 110, 345
char-ci>? ... 110, 345
char-code ... 113
char-code-limit ... 113
char-comparator ... 360
char-downcase ... 111, 345
char-foldcase ... 111, 345
char-general-category ... 114
char-graphic? ... 248
char-hash ... 355
char-in ... 133
char-in-set? ... 115
char-lower-case? ... 111, 345
char-not-in ... 133
char-numeric? ... 110, 345
char-predicate ... 113
char-ready? ... 235, 345
char-ready? on textual input port ... 251

char-set ... 115, 260, 350
char-set* ... 115
char-set->code-points ... 116
char-set->list ... 350
char-set->sre ... 353
char-set->string ... 350
char-set-adjoin ... 350
char-set-adjoin! ... 350
char-set-any ... 350
char-set-complement ... 350
char-set-complement! ... 350
char-set-contains? ... 350
char-set-copy ... 350
char-set-count ... 350
char-set-cursor ... 350
char-set-cursor-next ... 350
char-set-delete ... 350
char-set-delete! ... 350
char-set-diff+intersection ... 350
char-set-diff+intersection! ... 350
char-set-difference ... 116, 350
char-set-difference! ... 350
char-set-every ... 350
char-set-filter ... 350
char-set-filter! ... 350
char-set-fold ... 350
char-set-for-each ... 350
char-set-hash ... 350
char-set-intersection ... 116, 350
char-set-intersection! ... 350
char-set-intersection* ... 116
char-set-invert ... 116
char-set-map ... 350
char-set-predicate ... 115
char-set-ref ... 350
char-set-size ... 350
char-set-unfold ... 350
char-set-unfold! ... 350
char-set-union ... 116, 350
char-set-union! ... 350
char-set-union* ... 116
char-set-xor ... 350
char-set-xor! ... 350
char-set:alphabetic ... 116
char-set:alphanumeric ... 116
char-set:ascii ... 350
char-set:blank ... 350
char-set:digit ... 350
char-set:empty ... 350
char-set:full ... 350
char-set:graphic ... 350
char-set:hex-digit ... 350
char-set:iso-control ... 350
char-set:letter ... 350
char-set:letter+digit ... 350
char-set:lower-case ... 116, 350
char-set:numeric ... 116
char-set:printing ... 350
char-set:punctuation ... 350
char-set:symbol ... 350
char-set:title-case ... 350
char-set:upper-case ... 116, 350
char-set:whitespace ... 116, 129, 350
char-set<= ... 350
char-set= ... 350
char-set=? ... 116
char-set? ... 115, 350
char-title-case? ... 355
char-titlecase ... 355
char-upcase ... 111, 345
char-upper-case? ... 111, 345
char-whitespace? ... 110, 128, 345
char<=? ... 110, 345
char<? ... 110, 345
char=? ... 57, 110, 345
char>=? ... 110, 345
char>? ... 110, 345
char? ... 110, 345
chars-remaining on textual input port ... 252
circular-generator ... 359
circular-list ... 154, 347
circular-list? ... 144, 347
clear ... 240
clear-bit ... 86
close-all-open-files ... 231
close-input-port ... 229, 345
close-output-port ... 229, 345
close-port ... 229, 345
close-syntax ... 45
close-tcp-server-socket ... 306
code-point-general-category ... 114
code-point-in-set? ... 115
command-line ... 345
comparator-check-type ... 355
comparator-equality-predicate ... 355
comparator-hash ... 355
comparator-hash-function ... 355
comparator-hashable? ... 355
comparator-if<=> ... 355
comparator-max ... 360
comparator-max-in-list ... 360
comparator-min ... 360
comparator-min-in-list ... 360
comparator-ordered? ... 355
comparator-ordering-predicate ... 355
comparator-register-default! ... 355
comparator-test-type ... 355
comparator-type-test-predicate ... 355
comparator? ... 355
compile-regsexp ... 137
compiled-procedure? ... 216
complex? ... 66, 345
compound ... 81
compound-procedure? ... 216
compoundm1 ... 81
compute-char-set ... 115

conc-name 34
concatenate 347
concatenate! 347
cond 5, 26, 29, 167, 345
cond-expand 52, 345
condition-accessor 321, 323
condition-constructor 321
condition-predicate 321, 322
condition-signaller 321, 322
condition-type/error? 323
condition-type/field-names 324
condition-type/generalizations 324
condition-type:arithmetic-error 330
condition-type:bad-range-argument 68, 166, 197, 327
condition-type:breakpoint 310, 332
condition-type:control-error 331
condition-type:datum-out-of-range 327
condition-type:derived-file-error 329
condition-type:derived-port-error 329
condition-type:divide-by-zero 330
condition-type:error 313, 322, 323, 325
condition-type:file-error 328
condition-type:file-operation-error 284, 285, 286, 328
condition-type:floating-point-overflow ... 330
condition-type:floating-point-underflow 330
condition-type:illegal-datum 325
condition-type:inapplicable-object 328
condition-type:macro-binding 223
condition-type:no-such-restart 318, 331
condition-type:not-loading 331
condition-type:port-error 329
condition-type:primitive-procedure-error 330
condition-type:serious-condition 310, 325
condition-type:simple-condition 310, 332
condition-type:simple-error 311, 325
condition-type:simple-warning 311, 332
condition-type:subprocess-abnormal-termination 302
condition-type:subprocess-signalled 301
condition-type:subprocess-stopped 301
condition-type:system-call-error 331
condition-type:unassigned-variable 6, 223, 329
condition-type:unbound-variable 6, 22, 329
condition-type:variable-error 329
condition-type:warning 310, 331
condition-type:wrong-number-of-arguments 15, 216, 326
condition-type:wrong-type-argument 6, 326
condition-type:wrong-type-datum 326
condition-type? 324
condition/continuation 322
condition/error? 322
condition/report-string 323
condition/restarts 321, 322, 323
condition/type 322
condition? 322
conjugate 81
cons 140, 345, 347
cons* 143, 347
cons-stream 176
console-i/o-port 229
constructor 33
continuation? 219
continue 315, 318
copier 33
copy-area on x-graphics-device 343
copy-file 285
copysign 69
cos 74, 345
cos-pi* 80
count 347
count-accumulator 359
count-pairs 144
cpu-architecture 352
create-image on graphics-device 338
current-error-port 228, 345
current-file-time 289
current-input-port 228, 230, 345
current-jiffy 345
current-output-port 228, 230, 345
current-parser-macros 266
current-second 345

D

day-of-week/long-string 298
day-of-week/short-string 298
debug 248
decoded-time->file-time 296
decoded-time->string 296
decoded-time->universal-time 295
decoded-time/date-string 298
decoded-time/day 293
decoded-time/day-of-week 294
decoded-time/daylight-savings-time? 294
decoded-time/hour 293
decoded-time/minute 293
decoded-time/month 293
decoded-time/second 293
decoded-time/time-string 298
decoded-time/year 293
decoded-time/zone 294
default-comparator 360
default-hash 355
default-object? 15
default-random-source 106, 351
define 7, 22, 23, 29, 225, 345, 360
define-*matcher-expander 265
define-*matcher-macro 265
define-*parser-expander 265
define-*parser-macro 265

define-print-method 246
define-record-type 55, 345, 350, 356
define-structure 31, 172
define-syntax 39, 345
define-values 345
del-assoc 182
del-assoc! 183
del-assq 182
del-assq! 183
del-assv 182
del-assv! 183
delay 18, 174, 345
delay-force 345
delete 147, 347
delete! 147, 148, 347
delete-association-procedure 183
delete-directory 286
delete-duplicates 347
delete-duplicates! 347
delete-file 286, 345
delete-file-no-errors 286
delete-member-procedure 148
delq 148
delq! 148
delv 148
delv! 148
denominator 72, 345
digit->char 112
digit-value 111, 345
directory-namestring 282
directory-pathname 282
directory-pathname-as-file 283
directory-pathname? 280
directory-read 291
discard-chars on textual input port 251
discard-matched 260, 263
discard-parser-buffer-head! 256
discretionary-flush-output 238
discretionary-flush-output on textual output port 253
display 239, 312, 345
do 7, 29, 30, 345
dotted-list? 144, 347
draw-arc 341
draw-arc on x-graphics-device 341
draw-circle 342
draw-circle on x-graphics-device 342
draw-image on graphics-device 339
draw-subimage on graphics-device 339
drop 145, 347
drop-right 347
drop-right! 347
drop-while 347
dynamic-wind 219, 345

E

eighth 145, 347
else 5, 26, 27, 345
emergency-exit 345
encapsulate 264
end-of-char-set? 350
end-of-input 260
enough-namestring 282
enough-pathname 282
entity-extra 221
entity-procedure 221
entity? 221
enumerate-graphics-types 333
environment 304, 345
environment-assign! 224
environment-assignable? 224
environment-assigned? 224
environment-bindings 223
environment-bound-names 223
environment-bound? 224
environment-definable? 224
environment-define 224
environment-define-macro 225
environment-has-parent? 223
environment-lookup 224
environment-lookup-macro 224
environment-macro-names 223
environment-parent 223
environment-reference-type 224
environment? 223
eof-object 235, 345
eof-object? 235, 345
eof? on textual input port 252
ephemeron-broken? 179, 353
ephemeron-datum 179, 353
ephemeron-key 179, 353
ephemeron? 179, 353
epoch 292
eq-comparator 360
eq-hash 190, 196
eq-hash-mod 196
eq? .. 57, 60, 168, 170, 182, 183, 184, 186, 190, 197, 320, 345
equal-comparator 360
equal-hash 190, 196
equal-hash-mod 196
equal? 25, 57, 61, 182, 183, 187, 190, 345
eqv-comparator 360
eqv-hash 196
eqv-hash-mod 196
eqv? 9, 27, 57, 140, 182, 183, 186, 187, 345
er-macro-transformer 49
error 4, 309, 311, 313, 314, 345, 351
error-irritant/noise 313
error-object-irritants 345
error-object-message 345
error-object? 345
error:bad-range-argument 327

error:datum-out-of-range ... 327
error:derived-file ... 329
error:derived-port ... 329
error:divide-by-zero ... 330
error:file-operation ... 328
error:no-such-restart ... 331
error:wrong-number-of-arguments ... 327
error:wrong-type-argument ... 326
error:wrong-type-datum ... 326
euclidean-quotient ... 71
euclidean-remainder ... 71
euclidean/ ... 69
eval ... 225, 345
even? ... 67, 345
every ... 152, 347
exact ... 64, 82, 345
exact->inexact ... 81
exact-integer-sqrt ... 345
exact-integer? ... 67, 345
exact-nonnegative-integer? ... 67
exact-rational? ... 67
exact? ... 67, 345
except-last-pair ... 147
except-last-pair! ... 147
exit ... 345
exp ... 74, 345
exp10 ... 76
exp10m1 ... 76
exp2 ... 76
exp2m1 ... 76
expm1 ... 75
export-random-state ... 105
expt ... 81, 345
exsec ... 76
exsec-pi* ... 80
extend-top-level-environment ... 226
extract-bit ... 86

F

false ... 167
false? ... 167
features ... 345
fifth ... 145, 347
file-access ... 288
file-access-time ... 289
file-access-time-direct ... 289
file-access-time-indirect ... 289
file-attributes ... 290
file-attributes-direct ... 290
file-attributes-indirect ... 290
file-attributes/access-time ... 290
file-attributes/change-time ... 290
file-attributes/gid ... 291
file-attributes/inode-number ... 291
file-attributes/length ... 290
file-attributes/mode-string ... 290
file-attributes/modes ... 291
file-attributes/modification-time ... 290
file-attributes/n-links ... 290
file-attributes/type ... 290
file-attributes/uid ... 290
file-directory? ... 287
file-eq? ... 288
file-error? ... 230, 231, 345
file-executable? ... 288
file-exists-direct? ... 285
file-exists-indirect? ... 285
file-exists? ... 285, 345
file-length ... 290
file-modes ... 288
file-modification-time ... 289
file-modification-time-direct ... 289
file-modification-time-indirect ... 289
file-namestring ... 282
file-pathname ... 282
file-readable? ... 288
file-regular? ... 287
file-symbolic-link? ... 287
file-time->global-decoded-time ... 296
file-time->global-time-string ... 297
file-time->local-decoded-time ... 296
file-time->local-time-string ... 297
file-time->universal-time ... 296
file-touch ... 290
file-type-direct ... 287
file-type-indirect ... 287
file-writeable? ... 288
fill-circle ... 342
fill-circle on x-graphics-device ... 342
fill-with ... 127
filter ... 147, 347
filter! ... 147, 347
filter-map ... 347
find ... 149, 347
find-restart ... 315, 319
find-tail ... 149, 347
finite? ... 345
first ... 145, 347
first-set-bit ... 86
fix:* ... 88
fix:+ ... 88
fix:- ... 88
fix:-1+ ... 88
fix:< ... 88
fix:<= ... 88
fix:= ... 88
fix:> ... 88
fix:>= ... 88
fix:1+ ... 88
fix:and ... 89
fix:andc ... 89
fix:divide ... 89
fix:fixnum? ... 88
fix:gcd ... 88
fix:lsh ... 89

fix:negative? ... 88
fix:not ... 89
fix:or ... 89
fix:positive? ... 88
fix:quotient ... 88
fix:remainder ... 88
fix:xor ... 89
fix:zero? ... 88
fixnum? ... 358
flo:* ... 92
flo:*+ ... 92
flo:*- ... 92
flo:+ ... 92
flo:- ... 92
flo:/ ... 92
flo:< ... 91
flo:<= ... 91
flo:<> ... 91
flo:= ... 91
flo:> ... 91
flo:>= ... 91
flo:abs ... 93
flo:acos ... 94
flo:acos/pi ... 94
flo:acosh ... 94
flo:aexsec ... 94
flo:aexsec/pi ... 94
flo:asin ... 94
flo:asin/pi ... 94
flo:asinh ... 94
flo:atan ... 94
flo:atan/pi ... 94
flo:atan2 ... 95
flo:atan2/pi ... 95
flo:atanh ... 94
flo:aversin ... 94
flo:aversin/pi ... 94
flo:cbrt ... 94
flo:ceiling ... 95
flo:ceiling->exact ... 95
flo:classify ... 92
flo:clear-exceptions! ... 101
flo:compound ... 94
flo:compoundm1 ... 94
flo:copysign ... 93
flo:cos ... 94
flo:cos-pi* ... 94
flo:cosh ... 94
flo:default-environment ... 99
flo:default-rounding-mode ... 103
flo:default-trapped-exceptions ... 102
flo:defer-exception-traps! ... 102
flo:deferring-exception-traps ... 103
flo:environment ... 99
flo:erf ... 94
flo:erfc ... 94
flo:error-bound ... 96
flo:exception:divide-by-zero ... 101
flo:exception:inexact-result ... 101
flo:exception:invalid-operation ... 101
flo:exception:overflow ... 101
flo:exception:subnormal-operand ... 101
flo:exception:underflow ... 101
flo:exceptions->names ... 101
flo:exp ... 93
flo:exp10 ... 93
flo:exp10m1 ... 93
flo:exp2 ... 93
flo:exp2m1 ... 93
flo:expm1 ... 93
flo:expt ... 94
flo:exsec ... 94
flo:exsec-pi* ... 94
flo:fast-fma? ... 93
flo:finite? ... 92
flo:flonum? ... 90
flo:floor ... 94
flo:floor->exact ... 95
flo:fma ... 92
flo:gamma ... 94
flo:greatest-normal-exponent-base-10 ... 97
flo:greatest-normal-exponent-base-2 ... 97
flo:greatest-normal-exponent-base-e ... 97
flo:have-trap-enable/disable? ... 102
flo:hypot ... 94
flo:ignoring-exception-traps ... 103
flo:infinite? ... 92
flo:j0 ... 94
flo:j1 ... 94
flo:jn ... 94
flo:largest-positive-normal ... 97
flo:ldexp ... 96
flo:least-normal-exponent-base-10 ... 97
flo:least-normal-exponent-base-2 ... 97
flo:least-normal-exponent-base-e ... 97
flo:least-subnormal-exponent-base-10 ... 97
flo:least-subnormal-exponent-base-2 ... 97
flo:least-subnormal-exponent-base-e ... 97
flo:lgamma ... 94
flo:log ... 93
flo:log-error-bound ... 96
flo:log-ulp-of-one ... 96
flo:log10 ... 93
flo:log10p1 ... 94
flo:log1p ... 93
flo:log2 ... 93
flo:log2p1 ... 94
flo:logb ... 96
flo:logp1 ... 93
flo:make-nan ... 98
flo:max ... 95
flo:max-mag ... 95
flo:max-mag-num ... 96
flo:max-num ... 95
flo:min ... 95
flo:min-mag ... 95

flo:min-mag-num 96
flo:min-num 95
flo:names->exceptions 101
flo:nan-payload 98
flo:nan-quiet? 98
flo:nan? 92
flo:negate 93
flo:negative? 91
flo:nextafter 96
flo:normal-exponent-max 97
flo:normal-exponent-min 97
flo:normal? 92
flo:positive? 91
flo:precision 96
flo:preserving-environment 99
flo:radix 96
flo:radix. 96
flo:raise-exceptions! 101
flo:random-unit 105
flo:random-unit-closed 104
flo:random-unit-open 104
flo:restore-exception-flags! 101
flo:round 95
flo:round->exact 95
flo:rounding-mode 104
flo:rounding-modes 103
flo:rsqrt 94
flo:safe-zero? 92
flo:safe< 91
flo:safe<= 91
flo:safe<> 91
flo:safe= 91
flo:safe> 91
flo:safe>= 91
flo:save-exception-flags 101
flo:scalbn 96
flo:set-environment! 99
flo:set-rounding-mode! 104
flo:set-trapped-exceptions! 102
flo:sign-negative? 92
flo:signed-lgamma 95
flo:sin 94
flo:sin-pi* 94
flo:sinh 94
flo:smallest-positive-normal 97
flo:smallest-positive-subnormal 97
flo:sqrt 94
flo:sqrt1pm1 94
flo:subnormal-exponent-min 97
flo:subnormal? 92
flo:supported-exceptions 101
flo:tan 94
flo:tan-pi* 94
flo:tanh 94
flo:test-exception-flags 101
flo:test-exceptions 101
flo:total-mag< 98
flo:total-order 98
flo:total-order-mag 98
flo:total< 98
flo:trap-exceptions! 102
flo:trappable-exceptions 101
flo:trapped-exceptions 102
flo:truncate 95
flo:truncate->exact 95
flo:ulp 97
flo:ulp-of-one 96
flo:unordered? 91
flo:untrap-exceptions! 102
flo:update-environment! 99
flo:versin 94
flo:versin-pi* 94
flo:with-default-environment 99
flo:with-exceptions-trapped 102
flo:with-exceptions-untrapped 102
flo:with-rounding-mode 104
flo:with-trapped-exceptions 102
flo:y0 94
flo:y1 94
flo:yn 94
flo:zero? 91
flonum-parser-fast? 82
flonum-unparser-cutoff 83
floor 73, 345
floor->exact 73
floor-quotient 71, 345
floor-remainder 71, 345
floor/ 69, 345
fluid-let 22
flush-output 240
flush-output on textual output port 253
flush-output-port 238, 240, 345
fold 152, 347
fold-left 153
fold-right 153, 347
font-structure on x-graphics-device 343
for-each 151, 345, 347
force 174, 177, 345
format 244
format-error-message 313
fourth 145, 347
fresh-line 240
fresh-line on textual output port 253
fx* 358
fx*/carry 358
fx+ 358
fx+/carry 358
fx- 358
fx-/carry 358
fx-greatest 358
fx-least 358
fx-width 358
fx<=? 358
fx<? 358
fx=? 358
fx>=? 358

fx>? ... 358
fxabs ... 358
fxand ... 358
fxarithmetic-shift ... 358
fxarithmetic-shift-left ... 358
fxarithmetic-shift-right ... 358
fxbit-count ... 358
fxbit-field ... 358
fxbit-field-reverse ... 358
fxbit-field-rotate ... 358
fxbit-set? ... 358
fxcopy-bit ... 358
fxeven? ... 358
fxfirst-set-bit ... 358
fxif ... 358
fxior ... 358
fxlength ... 358
fxmax ... 358
fxmin ... 358
fxneg ... 358
fxnegative? ... 358
fxnot ... 358
fxodd? ... 358
fxpositive? ... 358
fxquotient ... 358
fxremainder ... 358
fxsqrt ... 358
fxsquare ... 358
fxxor ... 358
fxzero? ... 358

G

gappend ... 359
gc-reclaimed-object ... 178
gc-reclaimed-object? ... 178
gcd ... 72, 345
gcombine ... 359
gcons* ... 359
gdelete ... 359
gdelete-neighbor-dups ... 359
gdrop ... 359
gdrop-while ... 359
ge ... 7, 226
general-car-cdr ... 142
generate-uninterned-symbol ... 170
generator ... 359
generator->list ... 359
generator->reverse-list ... 359
generator->string ... 359
generator->vector ... 359
generator->vector! ... 359
generator-any ... 359
generator-count ... 359
generator-every ... 359
generator-find ... 359
generator-fold ... 359
generator-for-each ... 359
generator-map->list ... 359
generator-unfold ... 359
get-default on x-graphics-device ... 343
get-environment-variable ... 345
get-environment-variables ... 345
get-host-by-address ... 307
get-host-by-name ... 307
get-host-name ... 307
get-output-bytevector ... 233, 345
get-output-string ... 231, 232, 345
get-parser-buffer-pointer ... 255
get-parser-buffer-tail ... 255
get-universal-time ... 292
gfilter ... 359
gflatten ... 359
ggroup ... 359
gindex ... 359
global-decoded-time ... 292
global-parser-macros ... 265
gmap ... 359
gmerge ... 359
grapheme-cluster-length ... 122
grapheme-cluster-slice ... 122
graphics-bind-drawing-mode ... 337
graphics-bind-line-style ... 337
graphics-clear ... 334, 341, 342
graphics-close ... 334
graphics-coordinate-limits ... 334
graphics-device-coordinate-limits ... 334
graphics-disable-buffering ... 337
graphics-drag-cursor ... 336
graphics-draw-line ... 335
graphics-draw-point ... 335
graphics-draw-text ... 335
graphics-enable-buffering ... 337
graphics-erase-point ... 335
graphics-flush ... 337
graphics-move-cursor ... 335
graphics-operation ... 338
graphics-reset-clip-rectangle ... 338
graphics-set-clip-rectangle ... 338
graphics-set-coordinate-limits ... 334
graphics-set-drawing-mode ... 337
graphics-set-line-style ... 337
graphics-type-available? ... 333
gremove ... 359
group ... 136
group-ref ... 136
gselect ... 359
gstate-filter ... 359
gtake ... 359
gtake-while ... 359
guarantee-procedure-of-arity ... 217
guard ... 345

H

hamming-distance 86
hard-link-file 286
hash 196, 197, 352, 354
hash-bound 355
hash-by-eqv 196
hash-by-identity 196, 352, 354
hash-object 197
hash-salt 355
hash-table 354
hash-table->alist 192, 352, 354
hash-table-clean! 193
hash-table-clear! 191, 354
hash-table-constructor 188
hash-table-contains? 354
hash-table-copy 352, 354
hash-table-count 354
hash-table-delete! 191, 352, 354
hash-table-difference! 354
hash-table-empty-copy 354
hash-table-empty? 354
hash-table-entries 354
hash-table-entry-type:datum-ephemeral 190
hash-table-entry-type:datum-weak 189
hash-table-entry-type:key&datum-ephemeral 190
hash-table-entry-type:key&datum-weak 189
hash-table-entry-type:key-ephemeral 189
hash-table-entry-type:key-weak 189
hash-table-entry-type:key/datum-weak 189
hash-table-entry-type:strong 189
hash-table-equivalence-function 352, 354
hash-table-exists? 352, 354
hash-table-find 354
hash-table-fold 352, 354
hash-table-for-each 354
hash-table-grow-size 195
hash-table-hash-function 352, 354
hash-table-intern! 193, 354
hash-table-intersection! 354
hash-table-keys 192, 352, 354
hash-table-map 354
hash-table-map! 354
hash-table-map->list 354
hash-table-merge! 352, 354
hash-table-mutable? 354
hash-table-pop! 354
hash-table-prune! 354
hash-table-ref 191, 352, 354
hash-table-ref/default 191, 352, 354
hash-table-rehash-size 195
hash-table-rehash-threshold 195
hash-table-set! 191, 352, 354
hash-table-shrink-size 195
hash-table-size 191, 352, 354
hash-table-unfold 354
hash-table-union! 354
hash-table-update! 192, 352, 354
hash-table-update!/default 193, 352, 354
hash-table-values 192, 352, 354
hash-table-walk 192, 352, 354
hash-table-xor! 354
hash-table/clean! 193
hash-table/clear! 191
hash-table/constructor 190
hash-table/count 191
hash-table/datum-list 192
hash-table/for-each 192
hash-table/get 191
hash-table/intern! 193
hash-table/key-list 192
hash-table/lookup 192
hash-table/make 197
hash-table/modify! 193
hash-table/put! 191
hash-table/rehash-size 195
hash-table/rehash-threshold 195
hash-table/remove! 191
hash-table/size 195
hash-table=? 354
hash-table? 191, 352, 354
host-address-any 308
host-address-loopback 308
host-namestring 282
host=? 283
host? 283

I

i/o-port-type? 250
i/o-port? 228
identifier=? 48
identifier? 48
if 26, 167, 345
ignore-error 314
ignore-errors 313
imag-part 81, 345
image/destroy 339
image/fill-from-byte-vector 339
image/height 339
image/width 339
image? 339
implementation-name 352
implementation-version 352
implemented-primitive-procedure? 218
import-random-state 105
include 345
include-ci 345
inexact 81, 345
inexact->exact 82
inexact? 67, 345
infinite? 345
init-file-pathname 283
initial-offset 35
input 302
input-buffer-size 302

input-buffer-size on textual input port ... 252
input-line-translation ... 302
input-port->parser-buffer ... 254
input-port-blocking-mode ... 242
input-port-open? ... 228, 345
input-port-terminal-mode ... 243
input-port-type? ... 250
input-port/char-ready? ... 252
input-port/discard-chars ... 252
input-port/peek-char ... 252
input-port/read-char ... 252
input-port/read-string ... 252
input-port/read-substring ... 252
input-port? ... 228, 345
integer->char ... 111, 345
integer-ceiling ... 71
integer-divide ... 72, 89
integer-divide-quotient ... 72, 89
integer-divide-remainder ... 72, 89
integer-floor ... 71
integer-length ... 86
integer-round ... 71
integer-truncate ... 71
integer? ... 66, 345
interaction-environment ... 345
interaction-i/o-port ... 228, 247
intern ... 169
intern-soft ... 169
internal-time/seconds->ticks ... 299
internal-time/ticks->seconds ... 299
interpreter-environment? ... 226
invoke-restart ... 315, 316, 319, 320
invoke-restart-interactively ... 315, 319, 320
iota ... 143, 347
istring? ... 357

J

jiffies-per-second ... 345

K

keyword-constructor ... 34

L

lambda ... 5, 7, 12, 14, 15, 18, 22, 23, 29, 215, 345
last ... 147, 347
last-pair ... 147, 347
lcm ... 72, 345
leading ... 127, 128
length ... 64, 144, 345, 347
length+ ... 144, 347
let ... 7, 17, 22, 23, 29, 345
let* ... 7, 17, 22, 23, 29, 345
let*-syntax ... 39
let*-values ... 20, 22, 23, 29, 345
let-syntax ... 37, 345
let-values ... 19, 22, 23, 29, 345
letrec ... 7, 18, 22, 23, 29, 345
letrec* ... 19, 22, 23, 29, 345
letrec-syntax ... 38, 345
line-end ... 133
line-start ... 133
link-variables ... 226
list ... 6, 142, 143, 154, 157, 232, 345, 347
list->char-set ... 350
list->char-set! ... 350
list->generator ... 359
list->stream ... 176
list->string ... 121, 345, 357
list->vector ... 143, 157, 345
list-accumulator ... 359
list-comparator ... 360
list-copy ... 143, 183, 345, 347
list-deletor ... 148, 183
list-deletor! ... 148, 183
list-head ... 146
list-index ... 347
list-ref ... 145, 345, 347
list-set! ... 345
list-tabulate ... 143, 347
list-tail ... 146, 345
list= ... 347
list? ... 144, 182, 184, 345
load ... 345
load-option ... 244, 258, 266
local-decoded-time ... 292
local-host ... 283
log ... 74, 345
log-logistic ... 79
log10 ... 76
log10p1 ... 76
log1mexp ... 76
log1p ... 75
log1pexp ... 76
log2 ... 76
log2p1 ... 76
logistic ... 77
logistic-1/2 ... 78
logit ... 77
logit-exp ... 79
logit1/2+ ... 78
logp1 ... 76
logsumexp ... 80
lset-adjoin ... 347
lset-diff+intersection ... 347
lset-diff+intersection! ... 347
lset-difference ... 347
lset-difference! ... 347
lset-intersection ... 347
lset-intersection! ... 347
lset-union ... 347
lset-union! ... 347
lset-xor ... 347
lset-xor! ... 347

lset<= 347
lset= 347

M

machine-name 352
magnitude 81, 345
make-1d-table 184
make-accumulator 359
make-amap 208
make-apply-hook 221
make-bit-string 163
make-bytevector 345
make-cell 172
make-char 112
make-circular-list 154
make-comparator 355
make-condition 310, 311, 323
make-condition-type 323
make-coroutine-generator 359
make-datum-weak-eq-hash-table 186
make-datum-weak-eqv-hash-table 187
make-decoded-time 293
make-default-comparator 355
make-directory 286
make-entity 221
make-ephemeron 179, 353
make-eq-comparator 355
make-eq-hash-table 186
make-equal-comparator 355
make-equal-hash-table 187
make-eqv-comparator 355
make-eqv-hash-table 186
make-for-each-generator 359
make-graphics-device 333
make-hash-table 187, 352, 354
make-hash-table* 188
make-hash-table-type 188
make-hash-table-type* 189
make-initialized-list 143
make-initialized-vector 158
make-iota-generator 359
make-key-ephemeral-eq-hash-table 186
make-key-ephemeral-eqv-hash-table 187
make-key-weak-eq-hash-table 186
make-key-weak-eqv-hash-table 186
make-list 143, 345, 347
make-list-comparator 355
make-object-hash-table 186
make-object-hasher 197
make-pair-comparator 355
make-parameter 171, 345, 352
make-parser-macros 265
make-pathname 279
make-polar 81, 345
make-port 251
make-port-type 250
make-primitive-procedure 217
make-procedure-arity 216
make-promise 345
make-random-source 106, 351
make-random-state 105
make-range-generator 359
make-rb-tree 198
make-record-type 172
make-rectangular 74, 81, 345
make-root-top-level-environment 226
make-settable-parameter 171
make-string 118, 345
make-string-hash-table 187
make-strong-eq-hash-table 186
make-strong-eqv-hash-table 186
make-symbol-hash-table 186
make-syntactic-closure 45
make-synthetic-identifier 49
make-textual-port 250
make-textual-port-type 249
make-top-level-environment 226
make-unfold-generator 359
make-unsettable-parameter 171
make-vector 157, 345
make-vector-comparator 355
make-weak-eq-hash-table 186
make-weak-eqv-hash-table 186
make-wt-tree 203
make-wt-tree-type 202
make-xml-!attlist 272
make-xml-!element 271
make-xml-!entity 272
make-xml-!notation 272
make-xml-declaration 271
make-xml-document 271
make-xml-dtd 271
make-xml-element 271
make-xml-external-id 272
make-xml-name 268
make-xml-nmtoken 270
make-xml-parameter-!entity 272
make-xml-processing-instructions 271
make-xml-qname 269
make-xml-unparsed-!entity 272
map 150, 264, 345, 347
map! 347
map* 150
map-in-order 347
map-window on x-graphics-device 342
match 262
match-parser-buffer-char 256
match-parser-buffer-char-ci 256
match-parser-buffer-char-ci-no-advance 256
match-parser-buffer-char-in-set 256
match-parser-buffer-char-in-set-no-advance 256
match-parser-buffer-char-no-advance 256
match-parser-buffer-not-char 256
match-parser-buffer-not-char-ci 256

match-parser-buffer-not-char-ci-no-advance ... 256
match-parser-buffer-not-char-no-advance ... 256
match-parser-buffer-string ... 256
match-parser-buffer-string-ci ... 256
match-parser-buffer-string-ci-no-advance ... 256
match-parser-buffer-string-no-advance ... 256
match-parser-buffer-substring ... 256
match-parser-buffer-substring-ci ... 256
match-parser-buffer-substring-ci-no-advance ... 256
match-parser-buffer-substring-no-advance ... 256
max ... 68, 345
measure-interval ... 300
member ... 149, 345, 347
member-procedure ... 149
memq ... 149, 345, 347
memv ... 149, 345, 347
merge-pathnames ... 229, 281
merge-sort ... 155
merge-sort! ... 161
microcode-id/operating-system ... 306
microcode-id/operating-system-name ... 306
microcode-id/operating-system-variant ... 306
min ... 68, 345
modexp ... 72
modulo ... 71, 345
month/long-string ... 298
month/max-days ... 294
month/short-string ... 298
move-window on x-graphics-device ... 343
muffle-warning ... 311, 315, 318

N

name->char ... 110
named ... 35
named-lambda ... 16, 22, 29
nan? ... 345
NaN (not a number) ... 90
nearest-repl/environment ... 225
negative? ... 67, 345
newline ... 239, 345
nil ... 167
ninth ... 145, 347
noise ... 262
not ... 167, 345
not-char ... 259
not-char-ci ... 259
not-pair? ... 347
notification-output-port ... 228, 314
nt-file-mode/archive ... 289
nt-file-mode/compressed ... 289
nt-file-mode/directory ... 289
nt-file-mode/hidden ... 289
nt-file-mode/normal ... 289
nt-file-mode/read-only ... 289
nt-file-mode/system ... 289
nt-file-mode/temporary ... 289
null-list? ... 145, 347
null-xml-name-prefix ... 270
null-xml-name-prefix? ... 270
null-xml-namespace-uri ... 270
null-xml-namespace-uri? ... 270
null? ... 144, 145, 177, 345, 347
number->string ... 82, 345
number-hash ... 355
number-wt-type ... 202
number? ... 66, 345
numerator ... 72, 345

O

object-hash ... 197
object-hashed? ... 197
object-unhash ... 197
odd? ... 67, 345
open-binary-i/o-file ... 231
open-binary-input-file ... 230, 345
open-binary-output-file ... 230, 345
open-i/o-file ... 231
open-input-bytevector ... 233, 345
open-input-file ... 229, 230, 345
open-input-string ... 231, 345
open-output-bytevector ... 233, 345
open-output-file ... 229, 230, 345
open-output-string ... 231, 232, 345
open-tcp-server-socket ... 305
open-tcp-stream-socket ... 305
or ... 28, 167, 345
os-name ... 352
os-version ... 352
os/hostname ... 307
output ... 303
output-buffer-size ... 303
output-buffer-size on textual output port ... 253
output-line-translation ... 303
output-port-blocking-mode ... 242
output-port-open? ... 228, 345
output-port-terminal-mode ... 243
output-port-type? ... 250
output-port/discretionary-flush-output ... 253
output-port/flush-output ... 253
output-port/fresh-line ... 253
output-port/write-char ... 253
output-port/write-string ... 253
output-port/write-substring ... 253
output-port/x-size ... 254
output-port/y-size ... 254
output-port? ... 228, 345

P

pair-comparator 360
pair-fold 347
pair-fold-right 347
pair-for-each 347
pair? 140, 144, 178, 345, 347
param:default-pathname-defaults 281
param:flonum-printer-cutoff 83
param:print-with-maximum-readability? 241
param:printer-list-breadth-limit 240
param:printer-list-depth-limit 241
param:printer-radix 240
param:printer-string-length-limit 241
param:reader-fold-case? 238
param:reader-radix 237
parameterize ... 20, 22, 23, 29, 229, 230, 240, 345, 352
parameterize* 172
parse-namestring 276
parser-buffer-pointer-index 257
parser-buffer-pointer-line 257
parser-buffer-pointer? 255
parser-buffer-position-string 257
parser-buffer-ref 255
parser-buffer? 255
parser-macros? 265
partition 147, 347
partition! 147, 347
pathname-absolute? 280
pathname-as-directory 283, 285
pathname-default 282
pathname-default-device 280
pathname-default-directory 280
pathname-default-name 280
pathname-default-type 280
pathname-default-version 280
pathname-device 279
pathname-directory 279
pathname-host 279
pathname-name 279
pathname-new-device 279
pathname-new-directory 279
pathname-new-name 279
pathname-new-type 280
pathname-new-version 280
pathname-simplify 276
pathname-type 279
pathname-version 279
pathname-wild? 281
pathname=? 280
pathname? 280
peek-char 234, 345
peek-char on textual input port 251
peek-parser-buffer-char 255
peek-u8 ..... 236, 345
port-type/operation 250
port-type/operation-names 250
port-type/operations 250
port-type? 250
port/input-blocking-mode 242
port/input-terminal-mode 244
port/operation 251
port/operation-names 251
port/output-blocking-mode 242
port/output-terminal-mode 244
port/set-input-blocking-mode 242
port/set-input-terminal-mode 244
port/set-output-blocking-mode 242
port/set-output-terminal-mode 244
port/state 251
port/type 251
port/with-input-blocking-mode 242
port/with-input-terminal-mode 244
port/with-output-blocking-mode 242
port/with-output-terminal-mode 244
port? 228, 345
positive? 67, 345
pp 240
predicate 33
primitive-procedure-name 217
primitive-procedure? 216
print-procedure 33
procedure 167
procedure-arity 217
procedure-arity-max 216
procedure-arity-min 216
procedure-arity-valid? 217
procedure-arity? 216
procedure-environment 216
procedure-of-arity? 217
procedure? 215, 345
process-time-clock 299
product-accumulator 359
promise-forced? 175
promise-value 175
promise? 175, 345
prompt-for-command-char 248
prompt-for-command-expression 247
prompt-for-confirmation 248
prompt-for-evaluated-expression 248
prompt-for-expression 248
proper-list? 144, 347
pwd 284

Q

quasiquote 25, 139, 345
quick-sort 155
quick-sort! 161
quote 24, 139, 345
quotient 71, 345

R

raise ... 345
raise-continuable ... 345
random ... 104
random-bytevector ... 105
random-bytevector! ... 105
random-integer ... 106, 351
random-real ... 106, 351
random-source-make-integers ... 106, 351
random-source-make-reals ... 107, 351
random-source-pseudo-randomize! ... 106, 351
random-source-randomize! ... 106, 351
random-source-state-ref ... 106, 351
random-source-state-set! ... 106, 351
random-source? ... 106, 351
random-state? ... 105
rational? ... 66, 345
rationalize ... 73, 345
rationalize->exact ... 73
rb-tree->alist ... 199
rb-tree/copy ... 200
rb-tree/datum-list ... 199
rb-tree/delete! ... 199
rb-tree/delete-max! ... 201
rb-tree/delete-max-datum! ... 201
rb-tree/delete-max-pair! ... 201
rb-tree/delete-min! ... 201
rb-tree/delete-min-datum! ... 201
rb-tree/delete-min-pair! ... 201
rb-tree/empty? ... 199
rb-tree/equal? ... 199
rb-tree/height ... 200
rb-tree/insert! ... 198
rb-tree/intern! ... 199
rb-tree/key-list ... 199
rb-tree/lookup ... 198
rb-tree/max ... 201
rb-tree/max-datum ... 201
rb-tree/max-pair ... 201
rb-tree/min ... 200
rb-tree/min-datum ... 200
rb-tree/min-pair ... 200
rb-tree/ref ... 198
rb-tree/size ... 200
rb-tree/update! ... 199
rb-tree? ... 198
read ... 4, 8, 139, 168, 227, 233, 345
read-bytevector ... 236, 345
read-bytevector! ... 236, 345
read-char ... 227, 234, 235, 237, 345
read-char on textual input port ... 251
read-char-no-hang ... 234
read-delimited-string ... 237
read-error? ... 345
read-line ... 235, 345
read-only ... 32
read-parser-buffer-char ... 255
read-string ... 235, 345
read-string on textual input port ... 251
read-string! ... 235
read-substring on textual input port ... 251
read-substring! ... 236
read-u8 ... 227, 236, 345
read-xml ... 267
read-xml-file ... 267
real-comparator ... 360
real-part ... 81, 345
real-time-clock ... 299
real? ... 66, 345
receive ... 53, 349
record-accessor ... 173
record-constructor ... 172
record-keyword-constructor ... 173
record-modifier ... 173
record-predicate ... 173
record-type-descriptor ... 173
record-type-field-names ... 174
record-type-name ... 174
record-type? ... 174
record? ... 173
redisplay-hook ... 303
reduce ... 153, 347
reduce-left ... 154
reduce-right ... 154, 347
reference-barrier ... 180, 353
regexp ... 353
regexp-extract ... 353
regexp-fold ... 353
regexp-match->list ... 353
regexp-match-count ... 353
regexp-match-submatch ... 353
regexp-match-submatch-end ... 353
regexp-match-submatch-start ... 353
regexp-match? ... 353
regexp-matches ... 353
regexp-matches? ... 353
regexp-partition ... 353
regexp-replace ... 353
regexp-replace-all ... 353
regexp-search ... 353
regexp-split ... 353
regexp? ... 353
regsexp-match-string ... 137
regsexp-search-string-forward ... 137
remainder ... 71, 345
remove ... 147, 347
remove! ... 147, 347
rename-file ... 286
resize-window on x-graphics-device ... 343
restart/effector ... 320
restart/interactor ... 319, 320
restart/name ... 320
restart? ... 320
retry ... 315, 318
reverse ... 155, 345, 347
reverse! ... 155, 347

reverse-list->string 357
reverse-list->vector 356
reverse-list-accumulator 359
reverse-vector->generator 359
reverse-vector->list 356
reverse-vector-accumulator 359
round 73, 345
round->exact 73
round-quotient 71
round-remainder 71
round/ 69
rsc-macro-transformer 44
rsqrt 80
run-shell-command 301
run-synchronous-subprocess 301
runtime 299
rx 353

S

safe-accessors 35
sc-macro-transformer 43
scheme-subprocess-environment 304
second 145, 347
seq 136, 260, 263
set! 23, 24, 345
set-apply-hook-extra! 221
set-apply-hook-procedure! 221
set-background-color on
x-graphics-device 341
set-bit 86
set-border-color on x-graphics-device 341
set-border-width on x-graphics-device 342
set-car! 141, 345, 347
set-cdr! 139, 141, 232, 345, 347
set-cell-contents! 172
set-current-input-port! 229
set-current-output-port! 229
set-current-parser-macros! 266
set-entity-extra! 221
set-entity-procedure! 221
set-ephemeron-datum! 180
set-ephemeron-key! 180
set-file-modes! 288
set-file-times! 289
set-font on x-graphics-device 342
set-foreground-color on
x-graphics-device 341
set-hash-table-rehash-size! 195
set-hash-table-rehash-threshold! 195
set-hash-table/rehash-size! 195
set-hash-table/rehash-threshold! 195
set-input-buffer-size on
textual input port 252
set-input-port-blocking-mode! 242
set-input-port-terminal-mode! 243
set-interaction-i/o-port! 229
set-internal-border-width on
x-graphics-device 342
set-mouse-color on x-graphics-device 341
set-mouse-shape on x-graphics-device 342
set-notification-output-port! 229
set-output-buffer-size on
textual output port 253
set-output-port-blocking-mode 242
set-output-port-terminal-mode! 243
set-parser-buffer-pointer! 255
set-port/state! 251
set-record-type-unparser-method! 247
set-textual-port-state! 250
set-trace-output-port! 229
set-working-directory-pathname! 285
set-xml-!attlist-definitions! 272
set-xml-!attlist-name! 272
set-xml-!element-content-type! 271
set-xml-!element-name! 271
set-xml-!entity-name! 272
set-xml-!entity-value! 272
set-xml-!notation-id! 272
set-xml-!notation-name! 272
set-xml-declaration-encoding! 271
set-xml-declaration-standalone! 271
set-xml-declaration-version! 271
set-xml-document-declaration! 271
set-xml-document-dtd! 271
set-xml-document-misc-1! 271
set-xml-document-misc-2! 271
set-xml-document-misc-3! 271
set-xml-document-root! 271
set-xml-dtd-external! 271
set-xml-dtd-internal! 271
set-xml-dtd-root! 271
set-xml-element-attributes! 271
set-xml-element-content! 271
set-xml-element-name! 271
set-xml-external-id-id! 272
set-xml-external-id-uri! 272
set-xml-parameter-!entity-name! 272
set-xml-parameter-!entity-value! 272
set-xml-processing-instructions-name! 271
set-xml-processing-instructions-text! 271
set-xml-unparsed-!entity-id! 272
set-xml-unparsed-!entity-name! 272
set-xml-unparsed-!entity-notation! 272
seventh 145, 347
sexp 261, 264
shell-file-name 304
shift-left 85
shift-right 85
shiftin 87
shiftout 87
signal-condition 310, 311, 314
signed-integer->bit-string 166
simplest-exact-rational 74
simplest-rational 74

sin ... 74, 345
sin-pi* ... 80
singleton-wt-tree ... 203
sixth ... 145, 347
soft-link-file ... 286
sort ... 155
sort! ... 161
source->parser-buffer ... 255
span ... 347
span! ... 347
split-at ... 347
split-at! ... 347
sqrt ... 65, 80, 345
sqrt1pm1 ... 80
square ... 345
standard-error-handler ... 314
standard-error-hook ... 314
standard-print-method ... 246
standard-unparser-method ... 247
standard-warning-handler ... 314, 315
standard-warning-hook ... 315
store-value ... 315, 318
stream ... 176
stream->list ... 176
stream-car ... 177
stream-cdr ... 177
stream-first ... 177
stream-head ... 177
stream-length ... 177
stream-map ... 177
stream-null? ... 177
stream-pair? ... 176
stream-ref ... 177
stream-rest ... 177
stream-tail ... 177
string ... 118, 260, 345, 357
string* ... 118
string->char-set ... 350
string->char-set! ... 350
string->decoded-time ... 297
string->file-time ... 297
string->generator ... 359
string->list ... 121, 345, 357
string->nfc ... 122
string->nfd ... 122
string->number ... 84, 345
string->parser-buffer ... 254
string->symbol ... 169, 345
string->uninterned-symbol ... 170
string->universal-time ... 297
string->utf16 ... 357
string->utf16be ... 357
string->utf16le ... 357
string->utf8 ... 345, 357
string->vector ... 158, 345, 357
string->xml ... 267
string-accumulator ... 359
string-any ... 123, 357
string-append ... 121, 345, 357
string-append* ... 121
string-builder ... 124
string-ci ... 132, 260
string-ci-comparator ... 360
string-ci-hash ... 352, 354, 355
string-ci<=? ... 119, 345, 357
string-ci<? ... 119, 345, 357
string-ci=? ... 119, 124, 345, 357
string-ci>=? ... 119, 345, 357
string-ci>? ... 119, 345, 357
string-comparator ... 360
string-compare ... 119
string-compare-ci ... 119
string-concatenate ... 357
string-concatenate-reverse ... 357
string-contains ... 357
string-contains-right ... 357
string-copy ... 121, 345
string-copy! ... 121, 345
string-count ... 123, 357
string-downcase ... 120, 345, 357
string-drop ... 357
string-drop-right ... 357
string-end ... 133
string-every ... 124, 357
string-fill! ... 121, 345
string-filter ... 357
string-find-first-index ... 130
string-find-last-index ... 130
string-find-next-char ... 130
string-find-next-char-ci ... 130
string-find-next-char-in-set ... 130
string-find-previous-char ... 131
string-find-previous-char-ci ... 131
string-find-previous-char-in-set ... 131
string-fold ... 357
string-fold-right ... 357
string-foldcase ... 120, 345, 357
string-for-each ... 123, 345, 357
string-for-each-index ... 357
string-hash ... 124, 171, 190, 352, 354, 355
string-hash-ci ... 124
string-head ... 124
string-in-nfc? ... 122
string-in-nfd? ... 122
string-index ... 357
string-index-right ... 357
string-is-xml-name? ... 270
string-is-xml-nmtoken? ... 270
string-join ... 357
string-joiner ... 125
string-joiner* ... 125
string-length ... 64, 118, 345, 357
string-lower-case? ... 120
string-map ... 123, 345, 357
string-map-index ... 357
string-match-backward ... 131

string-match-forward 131
string-null? 124, 357
string-pad 357
string-pad-left 127
string-pad-right 127, 357
string-padder 126
string-prefix-ci? 131
string-prefix-length 357
string-prefix? 131, 357
string-ref 9, 118, 345, 357
string-remove 357
string-repeat 357
string-replace 129, 357
string-search-all 130
string-search-backward 129
string-search-forward 129
string-set! 9, 118, 169, 345
string-skip 357
string-skip-right 357
string-slice 120
string-split 357
string-splitter 126
string-start 133
string-suffix-ci? 132
string-suffix-length 357
string-suffix? 132, 357
string-tabulate 357
string-tail 124
string-take 357
string-take-right 357
string-titlecase 120, 355, 357
string-trim 129, 357
string-trim-both 357
string-trim-left 129
string-trim-right 129, 357
string-trimmer 128
string-unfold 357
string-unfold-right 357
string-upcase 120, 345, 357
string-upper-case? 120
string-word-breaks 122
string-wt-type 203
string<=? 119, 345, 357
string<? 119, 345, 357
string=? .. 57, 118, 124, 168, 169, 187, 190, 345, 357
string>=? 119, 345, 357
string>? 119, 345, 357
string? 118, 345, 357
strong-hash-table/constructor 190
sublist 146
substring 120, 345, 357
substring->parser-buffer 254
substring? 130
subvector 160
subvector->list 143
subvector-fill! 160
subvector-move-left! 160
subvector-move-right! 160
sum-accumulator 359
symbol->string 9, 57, 169, 345
symbol-append 170
symbol-hash 171, 355
symbol-hash-mod 171
symbol<? 171
symbol=? 345
symbol? 169, 345
syntax-error 345
syntax-rules 40, 345
system-clock 299
system-global-environment 225
system-library-directory-pathname 284
system-library-pathname 284

T

t 167
take 145, 347
take! 347
take-right 347
take-while 347
take-while! 347
tan 74, 345
tan-pi* 80
tcp-server-connection-accept 306
template 5
temporary-directory-pathname 287
temporary-file-pathname 287
tenth 145, 347
textual-i/o-port-type? 250
textual-input-port->parser-buffer 254
textual-input-port-type? 250
textual-output-port-type? 250
textual-port-operation 250
textual-port-operation-names 251
textual-port-state 250
textual-port-type 250
textual-port-type? 250
textual-port? 228, 345
the-environment 226
third 145, 347
thunk? 217
time-zone->string 298
time-zone? 294
toggle-bit 86
top-level-environment? 226
trace-output-port 228
trailing 127, 128
transform 264
tree-copy 142
true 167
truncate 73, 345
truncate->exact 73
truncate-quotient 71, 345
truncate-remainder 71, 345
truncate/ 69, 345
type 32, 34

type-descriptor 34

U

u8-ready? 236, 345
ucs-range->char-set 350
ucs-range->char-set! 350
unbind-variable 226
unfold 347
unfold-right 347
unhash 197
unhash-object 197
unicode-char? 114
unicode-code-point? 114
unicode-scalar-value? 114
universal-time->file-time 295
universal-time->global-decoded-time 295
universal-time->global-time-string 295
universal-time->local-decoded-time 295
universal-time->local-time-string 295
unless 345
unparser/set-tagged-pair-method! 247
unparser/set-tagged-vector-method! 247
unquote 25, 139, 345
unquote-splicing 25, 139, 345
unread-char 234
unsigned-integer->bit-string 166
unzip1 347
unzip2 347
unzip3 347
unzip4 347
unzip5 347
use-pty? 304
use-value 315, 318
user-homedir-pathname 283
user-initial-environment 7, 225
utf16->string 357
utf16be->string 357
utf16le->string 357
utf8->string 345, 357

V

valid-hash-number? 197
valid-object-hash? 197
valid-sre? 353
values 220, 263, 345
vector 157, 345
vector->generator 359
vector->list 143, 157, 345
vector->string 158, 345, 357
vector-accumulator 359
vector-accumulator! 359
vector-any 356
vector-append 345
vector-append-subvectors 356
vector-binary-search 159, 356
vector-comparator 360
vector-concatenate 356
vector-copy 157, 345
vector-copy! 345
vector-count 356
vector-cumulate 356
vector-eighth 159
vector-empty? 356
vector-every 356
vector-fifth 159
vector-fill! 160, 345
vector-first 159
vector-fold 356
vector-fold-right 356
vector-for-each 158, 345
vector-fourth 159
vector-grow 158
vector-head 160
vector-index 356
vector-index-right 356
vector-length 64, 159, 345
vector-map 158, 345
vector-map! 356
vector-partition 356
vector-ref 9, 159, 345
vector-reverse! 356
vector-reverse-copy 356
vector-reverse-copy! 356
vector-second 159
vector-set! 159, 345
vector-seventh 159
vector-sixth 159
vector-skip 356
vector-skip-right 356
vector-swap! 356
vector-tail 160
vector-third 159
vector-unfold 356
vector-unfold! 356
vector-unfold-right 356
vector-unfold-right! 356
vector= 356
vector? 159, 345
versin 76
versin-pi* 80

W

warn 311, 318
weak-car 179
weak-cdr 179
weak-cons 178
weak-hash-table/constructor 191
weak-pair/car? 178
weak-pair? 178
weak-set-car! 179
weak-set-cdr! 179
when 345
where 248

with-current-parser-macros 266
with-current-unparser-state 247
with-exception-handler 345
with-input-from-binary-file 230
with-input-from-file 230, 345
with-input-from-port 229
with-input-from-string 232
with-input-port-blocking-mode 242
with-input-port-terminal-mode 243
with-interaction-i/o-port 229
with-notification-output-port 229
with-output-port-blocking-mode 242
with-output-port-terminal-mode 243
with-output-to-binary-file 230
with-output-to-file 230, 345
with-output-to-port 229
with-output-to-string 232
with-output-to-truncated-string 232
with-pointer 262, 264
with-restart 310, 315, 316, 319
with-simple-restart 315, 316, 319
with-timings 299
with-trace-output-port 229
with-working-directory-pathname 285
withdraw-window on x-graphics-device 342
within-continuation 219
working-directory 304
working-directory-pathname 284
write 8, 168, 227, 232, 238, 312, 345
write-bytevector 239, 345
write-char 6, 227, 239, 345
write-char on textual output port 252
write-condition-report 314, 322, 323
write-line 240
write-restart-report 320
write-shared 238, 345
write-simple 239, 345
write-string 239, 244, 345
write-substring 239
write-substring on textual output port 253
write-to-string 232
write-u8 ..... 227, 239, 345
write-xml 267
write-xml-file 268
wt-tree/add 203
wt-tree/add! 204
wt-tree/delete 204
wt-tree/delete! 204
wt-tree/delete-min 208
wt-tree/delete-min! 208
wt-tree/difference 205
wt-tree/empty? 203
wt-tree/fold 206
wt-tree/for-each 206
wt-tree/index 207
wt-tree/index-datum 207
wt-tree/index-pair 207
wt-tree/intersection 205
wt-tree/lookup 204
wt-tree/member? 204
wt-tree/min 207
wt-tree/min-datum 207
wt-tree/min-pair 207
wt-tree/rank 207
wt-tree/set-equal? 205
wt-tree/size 203
wt-tree/split< 204
wt-tree/split> 204
wt-tree/subset? 205
wt-tree/union 204
wt-tree/union-merge 206
wt-tree? 203

X

x-character-bounds/ascent 343
x-character-bounds/descent 343
x-character-bounds/lbearing 343
x-character-bounds/rbearing 343
x-character-bounds/width 343
x-close-all-displays 341
x-font-structure/all-chars-exist 343
x-font-structure/character-bounds 343
x-font-structure/default-char 343
x-font-structure/direction 343
x-font-structure/max-ascent 343
x-font-structure/max-bounds 343
x-font-structure/max-descent 343
x-font-structure/min-bounds 343
x-font-structure/name 343
x-font-structure/start-index 343
x-geometry-string 341
x-graphics/close-display 341
x-graphics/open-display 340
x-open-display 340
x-size on textual output port 254
xcons 140, 347
xml-!attlist 272
xml-!attlist-definitions 272
xml-!attlist-name 272
xml-!attlist? 272
xml-!element 271
xml-!element-content-type 271
xml-!element-name 271
xml-!element? 271
xml-!entity 272
xml-!entity-name 272
xml-!entity-value 272
xml-!entity? 272
xml-!notation 272
xml-!notation-id 272
xml-!notation-name 272
xml-!notation? 272
xml->string 268
xml->wide-string 268
xml-declaration 271

xml-declaration-encoding ... 271
xml-declaration-standalone ... 271
xml-declaration-version ... 271
xml-declaration? ... 271
xml-document ... 271
xml-document-declaration ... 271
xml-document-dtd ... 271
xml-document-misc-1 ... 271
xml-document-misc-2 ... 271
xml-document-misc-3 ... 271
xml-document-root ... 271
xml-document? ... 271
xml-dtd ... 271
xml-dtd-external ... 271
xml-dtd-internal ... 271
xml-dtd-root ... 271
xml-dtd? ... 271
xml-element ... 271
xml-element-attributes ... 271
xml-element-content ... 271
xml-element-name ... 271
xml-element? ... 271
xml-external-id ... 272
xml-external-id-id ... 272
xml-external-id-uri ... 272
xml-external-id? ... 272
xml-name->symbol ... 269
xml-name-local ... 269
xml-name-prefix ... 269
xml-name-string ... 269
xml-name-uri ... 269
xml-name=? ... 269
xml-name? ... 268
xml-nmtoken? ... 270
xml-parameter-!entity ... 272
xml-parameter-!entity-name ... 272
xml-parameter-!entity-value ... 272
xml-parameter-!entity? ... 272
xml-processing-instructions ... 271
xml-processing-instructions-name ... 271
xml-processing-instructions-text ... 271
xml-processing-instructions? ... 271
xml-qname-local ... 269
xml-qname-prefix ... 269
xml-qname? ... 269
xml-unparsed-!entity ... 272
xml-unparsed-!entity-id ... 272
xml-unparsed-!entity-name ... 272
xml-unparsed-!entity-notation ... 272
xml-unparsed-!entity? ... 272
xml-uri ... 270
xmlns-uri ... 270
xsubstring ... 357

Y

y-size ... 249
y-size on textual output port ... 254

Z

zero? ... 66, 67, 345
zip ... 347

Appendix C Concept Index

!

! in mutation procedure names 10

"

" as external representation 117

#

as format parameter 244
in external representation of number 66
#(as external representation 157
#* as external representation 163
#[as external representation 246
#\ as external representation 109
#| as external representation 11
#b as external representation 66
#d as external representation 66
#e as external representation 66
#f as external representation 167
#i as external representation 66
#o as external representation 66
#t as external representation 167
#x as external representation 66

'

' as external representation 24

(

(as external representation 139

)

) as external representation 139

,

, as external representation 25
,@ as external representation 25

—

-| notational convention 4
-ci, in string procedure name 117

.

. as external representation 139
... in entries 5

;

; as external representation 11

=

=> in cond clause 27
=> notational convention 4

?

? in predicate names 10

[

[in entries 5

]

] in entries 5

`

` as external representation 25

\

\ as escape character in string 117

1

1D table (defn) 183

A

absolute pathname (defn) 284
absolute value, of number 68
access time, of file 289
access, used with set! 24
addition, of numbers 68
address hashing 195
alias 42, 47
alist (defn) 181
alphabetic case, of interned symbol 168
alphabetic case-insensitivity of programs (defn) 10
anonymous syntactic keyword 37
apostrophe, as external representation 24
appending, of bit strings 164
appending, of lists 146
appending, of symbols 170
appending, to output file 230
application hook (defn) 215, 220
application, of procedure 215
apply hook (defn) 220
argument evaluation order 14
arity 216
ASCII character 112
assignment 24
association list (defn) 181
association table (defn) 184
asterisk, as external representation 163
attribute, of file 290

B

backquote, as external representation 25
backslash, as escape character in string 117
Backtracking, in parser language 257
balanced binary trees 197, 201
barrier, reference 180
bell, ringing on console 240
binary port (defn) 227
binary trees 197, 201
binary trees, as discrete maps 202
binary trees, as sets 202
binding expression (defn) 7
binding expression, dynamic 20
binding expression, lexical 17
binding, of variable 6
binding, syntactic keyword 223
binding, unassigned 223
binding, variable 223
bit string (defn) 163
bit string index (defn) 163
bit string length (defn) 163
bitless character 113
bitmaps, graphics 338
bitwise operations 85
bitwise-logical operations, on fixnums 89
block structure 17
blocking mode, of port 242
BOA constructor 31
BOA constructor (defn) 33
body, of special form (defn) 5
boolean object 8
boolean object (defn) 167
boolean object, equivalence predicate 167
bound variable (defn) 6
bound-restarts 321, 323
bracket, in entries 5
broken ephemeron 179
bucky bit, of character (defn) 112
bucky bit, prefix (defn) 109
buffering, of graphics output 337
buffering, of output 238
built-in procedure 215
bytevector, input and output ports 233

C

call by need evaluation (defn) 174
car field, of pair (defn) 139
case clause 27
case conversion, of character 111
case folding, of character 111
case sensitivity, of string operations 117
case, of interned symbol 168
case-insensitivity of programs (defn) 10
cdr field, of pair (defn) 139
cell (defn) 172
character (defn) 109
character bits (defn) 112
character code (defn) 112
character set 115
character, bitless 113
character, input from port 234
character, input from textual port 251
character, output to textual port 252
character, searching string for 129
characters, special, in programs 11
child, of environment (defn) 6
circles, drawing 341, 342
circular list 144, 154
circular structure 61
clause, of case expression 27
clause, of cond expression 26
clearing the console screen 240
client socket 305
clip rectangle, graphics (defn) 338
clipping, of graphics 338
closing environment, of procedure (defn) 15
closing, of file port 231
closing, of port 229
code point 113
code, of character (defn) 112
code-point list 115
code-point range 115
combination (defn) 14
comma, as external representation 25

comment, extended, in programs (defn) 11
comment, in programs (defn) 11
comparison predicate 50
comparison, for equivalence 57
comparison, of bit strings 165
comparison, of boolean objects 167
comparison, of characters 110
comparison, of numbers 67
comparison, of XML names 269
compiled, procedure type 215
component selection, of bit string 164
component selection, of cell 172
component selection, of character 113
component selection, of ephemeron 179
component selection, of list 144
component selection, of pair 140
component selection, of stream 177
component selection, of vector 159
component selection, of weak pair 179
components, of pathname 277
compound procedure 215
cond clause 26
condition (defn) 320
condition handler (defn) 313
condition instance (defn) 320
condition signalling (defn) 310
condition type 310, 323
conditional expression (defn) 26
console, clearing 240
console, port 229
console, ringing the bell 240
constant 9
constant expression (defn) 13
constant, and quasiquote 25
constant, and quote 24
construction, of bit string 163
construction, of cell 172
construction, of character 112
construction, of circular list 154
construction, of continuation 218
construction, of EOF object 235
construction, of ephemeron 179
construction, of hash table 185
construction, of list 142
construction, of pair 140
construction, of pathname 276, 279
construction, of procedure 15
construction, of promise 174
construction, of stream 176
construction, of symbols 169
construction, of textual port type 249
construction, of vector 157
construction, of weak pair 178
continuation 218
continuation, alternate invocation 219
continuation, and dynamic binding 21
control, bucky bit prefix (defn) 109
conventions for error messages 312
conventions, lexical 9
conventions, naming 10
conventions, notational 4
conversion, pathname to string 276, 282
cooked mode, of terminal port 243
coordinates, graphics 334
copying, of alist 183
copying, of bit string 163
copying, of file 285
copying, of tree 142
copying, of vector 157
current environment 226
current environment (defn) 7
current error port (defn) 228
current input port (defn) 228
current input port, rebinding 230
current interaction port (defn) 228
current notification port (defn) 228
current output port (defn) 228
current output port, rebinding 230
current tracing output port (defn) 228
current working directory 275
current working directory (defn) 284
cursor, graphics (defn) 335
custom operations, on graphics device 338
custom operations, on textual port 249
cutting, of bit string 164
cutting, of list 145
cutting, of vector 160

D

d, as exponent marker in number 66
decoded time 291
default environment, floating-point 99
default object (defn) 15
defaulting, of pathname 281
define, procedure (defn) 22
definition 22
definition, internal 23
definition, internal (defn) 22
definition, top-level 23
definition, top-level (defn) 22
deletion, of alist element 182
deletion, of file 286
deletion, of list element 147
delimiter, in programs (defn) 10
denormal 90
device coordinates, graphics (defn) 334
device, pathname component 277
difference, of numbers 68
directive, format (defn) 244
directory path (defn) 278
directory, converting pathname to 283
directory, current working (defn) 284
directory, pathname component 277
directory, predicate for 287
directory, reading 283, 291

discrete maps, using binary trees 202
discretionary flushing, of buffered output 238
disembodied property list 168
display, clearing 240
display, X graphics 340
divide-by-zero exception 100
division, of integers 69
division, of numbers 68
dot, as external representation 139
dotted notation, for pair (defn) 139
dotted pair (see pair) 139
double precision, of inexact number 66
double quote, as external representation 117
drawing arcs and circles, graphics 341, 342
drawing mode, graphics (defn) 336
dynamic binding 20, 314, 315
dynamic binding, and continuations 21
dynamic binding, versus static scoping 7
dynamic environment 20
dynamic extent 20
dynamic parameter (defn) 171
dynamic types (defn) 3

E

e, as exponent marker in number 66
effector, restart (defn) 315
element, of list (defn) 139
ellipsis, in entries 5
else clause, of case expression (defn) 27
else clause, of cond expression (defn) 26
empty list (defn) 139
empty list, external representation 139
empty list, predicate for 145
empty stream, predicate for 177
empty string, predicate for 124
end of file object (see EOF object) 235
end, of subvector (defn) 157
entity (defn) 220
entry format 5
environment (defn) 6
environment, current 226
environment, current (defn) 7
environment, extension (defn) 6
environment, initial (defn) 7
environment, interpreter 226
environment, of procedure 15
environment, procedure closing (defn) 15
environment, procedure invocation (defn) 15
environment, top-level 226
EOF object, construction 235
EOF object, predicate for 235
ephemerally held data, of hash table 186
ephemerally held keys, of hash table 186
ephemeron (defn) 179
ephemeron, broken 179
equality, of XML names 269
equivalence predicate (defn) 57
equivalence predicate, for bit strings 165
equivalence predicate, for boolean objects 167
equivalence predicate, for characters 110
equivalence predicate, for fixnums 88
equivalence predicate, for flonums 91
equivalence predicate, for numbers 67
equivalence predicate, for pathname host 283
equivalence predicate, for pathnames 280
equivalence predicate, of hash table 186
error messages, conventions 312
error port, current (defn) 228
error, in examples 4
error, unassigned variable 6
error, unbound variable (defn) 6
error–> notational convention 4
errors, notational conventions 4
escape character, for string 117
escape procedure (defn) 218
escape procedure, alternate invocation 219
evaluation order, of arguments 14
evaluation, call by need (defn) 174
evaluation, in examples 4
evaluation, lazy (defn) 174
evaluation, of s-expression 225
even number 67
exactness 64
examples 4
existence, testing of file 285
exit, non-local 218
explicit renaming 49
exponent marker (defn) 66
expression (defn) 12
expression, binding (defn) 7
expression, conditional (defn) 26
expression, constant (defn) 13
expression, input from port 233
expression, iteration (defn) 29
expression, literal (defn) 13
expression, procedure call (defn) 13
expression, special form (defn) 13
extended comment, in programs (defn) 11
extended real line 90
extension, of environment (defn) 6
extent, of dynamic binding (defn) 21
extent, of objects 3
external representation (defn) 8
external representation, and quasiquote 25
external representation, and quote 24
external representation, for bit string 163
external representation, for empty list 139
external representation, for list 139
external representation, for number 66
external representation, for pair 139
external representation, for procedure 215
external representation, for string 117
external representation, for symbol 168
external representation, for vector 157
external representation, parsing 233

extra object, of application hook 220

F

f, as exponent marker in number 66
false, boolean object 8
false, boolean object (defn) 167
false, in conditional expression (defn) 26
false, predicate for 167
FDL, GNU Free Documentation License 361
file (regular), predicate for 287
file name 275
file time 291
file type, procedure for 287
file, converting pathname directory to 283
file, end-of-file marker (see EOF object) 235
file, input and output ports 229
filename (defn) 275
filling, of bit string 165
filling, of vector 160
filtering, of list 147
fixnum (defn) 88
floating-point comparison, ordered 91
floating-point comparison, unordered 91
floating-point environment 98
floating-point environment, default 99
floating-point number, infinite 90
floating-point number, normal 90
floating-point number, not a number 90
floating-point number, subnormal 90
floating-point number, zero 90
flonum (defn) 90
flushing, of buffered output 238
folding, of list 152
forcing, of promise 174
form 42
form, special (defn) 13
formal parameter list, of lambda (defn) 15
format directive (defn) 244
format, entry 5

G

generalization, of condition types 310, 311, 323, 324
generalization, of condition types (defn) 309
gensym (see uninterned symbol) 170
geometry string, X graphics 341
grapheme cluster 121
graphics 333
graphics, bitmaps 338
graphics, buffering of output 337
graphics, clipping 338
graphics, coordinate systems 334
graphics, cursor (defn) 335
graphics, custom operations 338
graphics, device coordinates (defn) 334
graphics, drawing 334
graphics, drawing arcs and circles 341, 342
graphics, drawing mode (defn) 336
graphics, images 338
graphics, line style (defn) 336
graphics, opening and closing devices 333
graphics, output characteristics 336
graphics, virtual coordinates (defn) 334
greatest common divisor, of numbers 72
growing, of vector 158

H

handler, condition (defn) 313
hard linking, of file 286
hash table 185
hashing, of key in hash table 188
hashing, of object 196
hashing, of string 124
hashing, of symbol 171
home directory, as pathname 283
hook, application (defn) 215
host, in filename 275
host, pathname component 277
hostname, TCP 305
hygienic 37
hyper, bucky bit prefix (defn) 109

I

I/O, to bytevectors 233
I/O, to files 229
I/O, to strings 231
identifier 42
identifier (defn) 10
identity, additive 68
identity, multiplicative 68
images, graphics 338
immutable 9
immutable string 118
implementation restriction 64
implicit begin 29
improper list (defn) 139
index, of bit string (defn) 163
index, of list (defn) 145
index, of string (defn) 117
index, of subvector (defn) 157
index, of vector (defn) 157
inexact-result exception 100
infinity (`+inf.0`, `-inf.0`) 90
inheritance, of environment bindings (defn) 6
initial environment (defn) 7
initial size, of hash table 194
input 227
input form 43
input form, to macro 49
input operations 233
input port (defn) 227
input port, bytevector 233

input port, console ... 229
input port, current (defn) ... 228
input port, file ... 229
input port, string ... 231
input, XML ... 266
insensitivity, to case in programs (defn) ... 10
installed, as pathname component ... 279
instance, of condition (defn) ... 320
integer division ... 69
integer, converting to bit string ... 166
interaction port, current (defn) ... 228
interactive input ports (defn) ... 233
internal definition ... 23
internal definition (defn) ... 22
internal representation, for character ... 112
internal representation, for inexact number ... 66
interned symbol (defn) ... 168
interning, of symbols ... 169
interpreted, procedure type ... 215
interpreter environment ... 226
invalid-operation exception ... 90, 100
inverse, additive, of number ... 68
inverse, multiplicative, of number ... 68
inverse, of bit string ... 165
inverse, of boolean object ... 167
invocation environment, of procedure (defn) ... 15
iteration expression (defn) ... 29

J

joiner procedure, of strings ... 125
joining, of strings ... 125

K

key, of association list element (defn) ... 181
keyword ... 37
keyword binding ... 223
keyword constructor ... 31
keyword constructor (defn) ... 34
keyword, of special form (defn) ... 13

L

l, as exponent marker in number ... 66
lambda expression (defn) ... 15
lambda list (defn) ... 15
lambda, implicit in define ... 22
lambda, implicit in let ... 17
latent types (defn) ... 3
lazy evaluation (defn) ... 174
least common multiple, of numbers ... 72
length, of bit string ... 164
length, of bit string (defn) ... 163
length, of list (defn) ... 139
length, of stream ... 177
length, of string (defn) ... 117
length, of vector (defn) ... 157
letrec, implicit in define ... 23
lexical binding expression ... 17
lexical conventions ... 9
lexical scoping (defn) ... 7
library, system pathname ... 284
line style, graphics (defn) ... 336
linking (hard), of file ... 286
linking (soft), of file ... 286
list (defn) ... 139
list index (defn) ... 145
list, association (defn) ... 181
list, converting to stream ... 176
list, converting to vector ... 157
list, external representation ... 139
list, improper (defn) ... 139
literal expression (defn) ... 13
literal, and quasiquote ... 25
literal, and quote ... 24
literal, identifier as ... 10
local part, of XML name ... 269
location ... 9
location, of variable ... 6
log-odds ... 79, 80
log-probability ... 79, 80
logical operations, on fixnums ... 89
long precision, of inexact number ... 66
loopback interface ... 308
looping (see iteration expressions) ... 29
lowercase ... 10
lowercase, character conversion ... 111

M

macro ... 36
macro keyword ... 37
macro transformer ... 37, 43, 49
macro use ... 37
magnitude, of real number ... 68
manifest types (defn) ... 3
mapping, of list ... 150
mapping, of stream ... 177
mapping, of vector ... 158
Matcher language ... 259
Matcher procedure ... 259
matching, of strings ... 129
maximum, of numbers ... 68
memoization, of promise ... 174
merging, of pathnames ... 281
meta, bucky bit prefix (defn) ... 109
minimum, of numbers ... 68
modification time, of file ... 289
modification, of bit string ... 165
modification, of vector ... 160
modulus, of hashing procedure ... 188
modulus, of integers ... 69
moving, of bit string elements ... 165
moving, of vector elements ... 160
multiple values, from procedure ... 220

multiplication, of numbers 68
must be, notational convention 4
mutable 9
mutable string 118
mutation procedure (defn) 10

N

name, of file 286
name, of symbol 169
name, of value (defn) 6
name, pathname component 277
named lambda (defn) 16
named let (defn) 29
names, XML 268
naming conventions 10
NaN 90
negative infinity (`-inf.0`) 90
negative number 67
nesting, of quasiquote expressions 25
newest, as pathname component 279
NFC 122
NFD 122
non-local exit 218
normal floating-point number 90
Normalization Form C (NFC) 122
Normalization Form D (NFD) 122
not a number (NaN, `+nan.0`) 90
notation, dotted (defn) 139
notational conventions 4
notification port, current (defn) 228
null string, predicate for 124
number 63
number, external representation 66
number, pseudorandom generation 104
numeric precision, inexact 66
numerical input and output 82
numerical operations 66
numerical types 63

O

object hashing 196
odd number 67
oldest, as pathname component 279
one-dimensional table (defn) 183
operand, of procedure call (defn) 13
Operating-System Interface 275
operator, of procedure call (defn) 13
option, run-time-loadable 244, 258, 266
optional component, in entries 5
optional parameter (defn) 15
order, of argument evaluation 14
ordered comparison 91
ordering, of characters 110
ordering, of numbers 67
output 227
output form 43
output port (defn) 227
output port, bytevector 233
output port, console 229
output port, current (defn) 228
output port, file 229
output port, string 231
output procedures 238
output, XML 267
overflow exception 100

P

padder procedure 126
padding, of string 126
pair (defn) 139
pair, external representation 139
pair, weak (defn) 178
parameter list, of lambda (defn) 15
parameter, dynamic (defn) 171
parameter, entry category 5
parameter, optional (defn) 15
parameter, required (defn) 15
parameter, rest (defn) 15
parent, of directory 278
parent, of environment (defn) 6
parenthesis, as external representation 139, 157
Parser buffer 254
Parser language 257, 262
Parser procedure 262
parser, XML 266
Parser-buffer pointer 254
parsing, of external representation 233
pasting, of bit strings 164
pasting, of lists 145
pasting, of symbols 170
path, directory (defn) 278
pathname 275
pathname (defn) 275
pathname components 277
pathname, absolute (defn) 284
pathname, relative (defn) 284
period, as external representation 139
physical size, of hash table (defn) 193
pop count 86
port 227
port (defn) 227
port number, TCP 305
port, bytevector 233
port, console 229
port, current 228
port, file 229
port, string 231
positive infinity (`+inf.0`) 90
positive number 67
precision, of inexact number 66
predicate (defn) 10, 57
predicate, equivalence (defn) 57
prefix, of string 131

prefix, of XML name 269
pretty printer 240
primitive procedure (defn) 215
primitive, procedure type 215
print name, of symbol 169
printed output, in examples 4
procedure 215
procedure call (defn) 13
procedure define (defn) 22
procedure, closing environment (defn) 15
procedure, compiled 215
procedure, compound 215
procedure, construction 15
procedure, entry format 5
procedure, escape (defn) 218
procedure, interpreted 215
procedure, invocation environment (defn) 15
procedure, of application hook 220
procedure, primitive 215
procedure, type 215
product, of numbers 68
promise (defn) 174
promise, construction 174
promise, forcing 174
prompting 247
proper tail recursion (defn) 3
property list 181, 183, 184
property list, of symbol 168
protocol, restart (defn) 315
pseudorandom number generation 104

Q

qname, of XML name 268
quiet NaN 90, 100
quote, as external representation 24
quotient, of integers 69
quotient, of numbers 68
quoting 24

R

R7RS 3
random number generation 104
rational, simplest (defn) 73
raw mode, of terminal port 243
record-type descriptor (defn) 172
recursion (see tail recursion) 3
red-black binary trees 197
reference barrier 180
reference, strong (defn) 177
reference, variable (defn) 13
reference, weak (defn) 177
referentially transparent 37
region of variable binding, do 30
region of variable binding, internal definition ... 23
region of variable binding, lambda 15
region of variable binding, let 17
region of variable binding, let* 17
region of variable binding, letrec 18
region, of variable binding (defn) 7
regular file, predicate for 287
rehash size, of hash table (defn) 194
rehash threshold, of hash table (defn) 195
relative pathname (defn) 284
remainder, of integers 69
renaming procedure 49
renaming, of file 286
REP loop 311, 313, 314, 318
REP loop (defn) 7
REP loop, environment of 7
representation, external (defn) 8
required parameter (defn) 15
resizing, of hash table 193
resources, X graphics 340
rest parameter (defn) 15
restart (defn) 315
restart effector (defn) 315
restart protocol 315
restarts, bound 321, 323
result of evaluation, in examples 4
result, unspecified (defn) 4
reversal, of list 155
ringing the console bell 240
root, as pathname component 278
run-time-loadable option 244, 258, 266
runtime system 3

S

s, as exponent marker in number 66
s-expression 225
scalar value 113
scheme concepts 6
Scheme standard 3
scope (see region) 3
scoping, lexical (defn) 7
scoping, static 7
screen, clearing 240
searching, of alist 182
searching, of bit string 164
searching, of list 149
searching, of string 129
searching, of vector 159
selecting, of stream component 177
selection, components of pathname 282
selection, of bit string component 164
selection, of cell component 172
selection, of character component 113
selection, of ephemeron component 179
selection, of list component 144
selection, of pair component 140
selection, of vector component 159
selection, of weak pair component 179
semicolon, as external representation 11
sensitivity, to case in programs (defn) 10

sequencing expressions ... 28
server socket ... 305
service, TCP ... 305
set, of characters ... 115
sets, using binary trees ... 202
shadowing, of variable binding (defn) ... 6
short precision, of inexact number ... 66
sign bit, of number ... 69
signal an error (defn) ... 4
signalling NaN ... 90, 100
signalling, of condition (defn) ... 310
signed zero ... 90
simplest rational (defn) ... 73
simplification, of pathname ... 276
single precision, of inexact number ... 66
size, of hash table (defn) ... 193
slice, of string ... 120
socket ... 305
soft linking, of file ... 286
special characters, in programs ... 11
special form ... 15
special form (defn) ... 13
special form, entry category ... 5
specialization, of condition types ... 310, 311, 321, 322, 323
specialization, of condition types (defn) ... 309
specified result, in examples ... 4
splitter procedure ... 126
splitting, of string ... 126
SRFI 0 ... 52
SRFI 2 ... 54
SRFI 8 ... 53
SRFI 9 ... 54
SRFI syntax ... 51
standard operations, on textual port ... 249
standard Scheme (defn) ... 3
start, of subvector (defn) ... 157
static scoping ... 7
static scoping (defn) ... 3
static types (defn) ... 3
stream (defn) ... 176
stream, converting to list ... 176
string builder procedure ... 124
string index (defn) ... 117
string length (defn) ... 117
string slice ... 120
string, character (defn) ... 117
string, input and output ports ... 231
string, input from port ... 237
string, input from textual port ... 251
string, interning as symbol ... 169
string, of bits (defn) ... 163
string, searching string for ... 129
strong reference (defn) ... 177
strong types (defn) ... 3
strongly held data, of hash table ... 186
strongly held keys, of hash table ... 186
subnormal floating-point number ... 90
subnormal-operand exception ... 100
subprocess ... 300
substring, of bit string ... 164
substring, output to textual port ... 253
subtraction, of numbers ... 68
subvector (defn) ... 157
suffix, of string ... 132
sum, of numbers ... 68
super, bucky bit prefix (defn) ... 109
symbol (defn) ... 168
symbolic link, predicate for ... 287
symbolic linking, of file ... 286
synchronous subprocess ... 300
syntactic closure ... 43
syntactic closures ... 42
syntactic environment ... 42
syntactic keyword ... 14, 37
syntactic keyword (defn) ... 13
syntactic keyword binding ... 223
syntactic keyword, identifier as ... 10
synthetic identifier ... 42

T

table, association (defn) ... 184
table, one-dimensional (defn) ... 183
tail recursion (defn) ... 3
tail recursion, vs. iteration expression ... 29
taxonomical link, of condition type (defn) ... 309
terminal mode, of port ... 243
terminal screen, clearing ... 240
tetrachotomy ... 91
textual input port operations ... 251
textual output port operations ... 252
textual port (defn) ... 227
textual port primitives ... 249
textual port type ... 249
tick ... 298
time, decoded ... 291
time, file ... 291
time, string ... 291
time, universal ... 291
token, in programs (defn) ... 9
top-level definition ... 23
top-level definition (defn) ... 22
top-level environment ... 226
total ordering (defn) ... 155
tracing output port, current (defn) ... 228
transformer environment ... 43
tree, copying ... 142
trees, balanced binary ... 197, 201
trichotomy ... 91
trimmer procedure ... 128
trimming, of string ... 128
true, boolean object ... 8
true, boolean object (defn) ... 167
true, in conditional expression (defn) ... 26
truename, of input file ... 286

type predicate, for 1D table 184
type predicate, for alist 182
type predicate, for apply hook 221
type predicate, for bit string 164
type predicate, for boolean 167
type predicate, for cell 172
type predicate, for character 110
type predicate, for character set 115
type predicate, for compiled procedure 216
type predicate, for compound procedure 216
type predicate, for continuation 219
type predicate, for empty list 145
type predicate, for entity 221
type predicate, for environment 223
type predicate, for EOF object 235
type predicate, for ephemeron 179
type predicate, for fixnum 88
type predicate, for flonum 90
type predicate, for hash table 191
type predicate, for list 144
type predicate, for number 66
type predicate, for pair 140
type predicate, for pathname 280
type predicate, for pathname host 283
type predicate, for port 228
type predicate, for primitive procedure 216
type predicate, for procedure 215
type predicate, for promise 175
type predicate, for record 173
type predicate, for record type 174
type predicate, for stream pair 176
type predicate, for symbol 169
type predicate, for top-level environment 226
type predicate, for vector 159
type predicate, for weak pair 178
type, condition 310
type, of condition 323
type, of procedure 215
type, pathname component 277
types, latent (defn) 3
types, manifest (defn) 3

U

unassigned binding 223
unassigned variable 13
unassigned variable (defn) 6
unassigned variable, and assignment 24
unassigned variable, and definition 23
unassigned variable, and dynamic bindings 22
unassigned variable, and named let 29
unbound variable 13
unbound variable (defn) 6
underflow exception 100
Unicode 113
Unicode code point 113
Unicode normalization forms 122
Unicode scalar value 113
Uniform Resource Identifier 268
uninterned symbol (defn) 168
universal time 291
unordered comparison 91
unspecified result (defn) 4
up, as pathname component 278
uppercase 10
uppercase, character conversion 111
URI, of XML name 268
usable size, of hash table (defn) 193
usage environment 43

V

V as format parameter 244
valid index, of bit string (defn) 163
valid index, of list (defn) 145
valid index, of string (defn) 117
valid index, of subvector (defn) 157
valid index, of vector (defn) 157
value, of variable (defn) 6
values, multiple 220
variable binding 6, 223
variable binding, do 30
variable binding, fluid-let 22
variable binding, internal definition 23
variable binding, lambda 15
variable binding, let 17
variable binding, let* 17
variable binding, letrec 18
variable binding, top-level definition 23
variable reference (defn) 13
variable, adding to environment 22
variable, assigning values to 24
variable, binding region (defn) 7
variable, entry category 5
variable, identifier as 10
vector (defn) 157
vector index (defn) 157
vector length (defn) 157
vector, converting to list 143
version, pathname component 277
virtual coordinates, graphics (defn) 334

W

weak pair (defn) 178
weak pair, and 1D table 183
weak reference (defn) 177
weak types (defn) 3
weakly held data, of hash table 186
weakly held keys, of hash table 186
weight-balanced binary trees 201
whitespace, in programs (defn) 9
working directory (see current
working directory) 284

X

X display, graphics 340
X geometry string, graphics 341
X graphics 339
X resources, graphics 340
X window system 339
XML input 266
XML names 268
XML output 267
XML parser 266

Z

zero 67, 90